Formalization of Conflict Analysis of Programs with Procedures, Thread Creation, and Monitors in Isabelle/HOL

Peter Lammich
Markus Müller-Olm
Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
peter.lammich@uni-muenster.de and mmo@math.uni-muenster.de
September 19, 2015

Abstract

In this work we formally verify the soundness and precision of a static program analysis that detects conflicts (e.g. data races) in programs with procedures, thread creation and monitors with the Isabelle theorem prover. As common in static program analysis, our program model abstracts guarded branching by nondeterministic branching, but completely interprets the call-/return behavior of procedures, synchronization by monitors, and thread creation. The analysis is based on the observation that all conflicts already occur in a class of particularly restricted schedules. These restricted schedules are suited to constraint-system-based program analysis.

The formalization is based upon a flowgraph-based program model with an operational semantics as reference point.
1 Introduction

Conflicts are a common programming error in parallel programs. A conflict occurs if the same resource is accessed simultaneously by more than one process. Given a program $\pi$ and two sets of control points $U$ and $V$, the analysis problem is to decide whether there is an execution of $\pi$ that simultaneously reaches one control point from $U$ and one from $V$.

In this work, we use a flowgraph-based program model that extends a previously studied model [6] by reentrant monitors. In our model, programs can call recursive procedures, dynamically create new threads and synchronize via reentrant monitors. As usual in static program analysis, our program model abstracts away guarded branching by nondeterministic choice. We use an operational semantics as reference point for the correctness proofs. It models parallel execution by interleaving, i.e. just one thread is executed at any time and context switches may occur after every step. The next step is nondeterministically selected from all threads ready for execution. The analysis is based on a constraint system generated from the flowgraph. From its least solution, one can decide whether control points from $U$ and $V$ are simultaneously reachable or not.

It is notoriously hard to analyze concurrent programs with constraint systems because of the arbitrary fine-grained interleaving. The key idea behind our analysis is to use a restricted scheduling: While the interleaving semantics can switch the context after each step, the restricted scheduling just allows context switches at certain points of a thread’s execution. We can show that each conflict is also reachable under this restricted scheduling. The restricted schedules can be easily analyzed with constraint systems as most of the complexity generated by arbitrary interleaving does no longer occur due to the restrictions. The remaining concurrency effects can be smoothly handled by using the concept of acquisition histories [5].

Related Work In [6] we present a constraint-system-based analysis for programs with thread creation and procedures but without monitors. The abstraction from synchronization is common in this line of research: There are automata-based techniques [1, 2, 3] as well as constraint-system-based techniques [7, 6] to analyze programs with procedures and either parallel calls or thread creation, but without any synchronization. In [5, 4] analysis techniques for interprocedural parallel programs with a fixed number of initial threads and nested locks are presented. These nested locks are not syntactically bound to the program structure, but assumed to be well-nested, that is any unlock statement is required to release the lock that was acquired last by the thread. Moreover, there is no support for reentrant
We use monitors instead of locks. Monitors are syntactically bound to the program structure and thus well-nestedness is guaranteed statically. Additionally we directly support reentrant monitors. Our model cannot simulate well-nested locks where a lock statement and its corresponding unlock statement may be in different procedures (as in [5, 4]). As common programming languages like Java also use reentrant monitors rather than locks, we believe our model to be useful as well.

**Document structure**  This document contains a commented formalization of these ideas as a collection of Isabelle/HOL theories. A more abstract description is in preparation. This document starts with formalization monitor consistent interleaving (Section 2) and acquisition histories (Section 3). Labeled transition systems are formalized in Section 4, and Section 5 defines the notion of interleaving semantics. Flowgraphs are defined in Section 6, and Section 7 describes their operational semantics. Section 8 contains the formalization of the restricted interleaving and Section 9 contains the constraint systems. Finally, the main result of this development – the correctness of the constraint systems w.r.t. to the operational semantics – is briefly stated in Section 10.

## 2 Monitor Consistent Interleaving

```
theory ConsInterleave
imports Interleave Misc
begin

The monitor consistent interleaving operator is defined on two lists of arbitrary elements, provided an abstraction function \( \alpha \) that maps list elements to pairs of sets of monitors is available. \( \alpha e = (M, M') \) intuitively means that step \( e \) enters the monitors in \( M \) and passes (enters and leaves) the monitors in \( M' \). The consistent interleaving describes all interleavings of the two lists that are consistent w.r.t. the monitor usage.

### 2.1 Monitors of lists of monitor pairs

The following defines the set of all monitors that occur in a list of pairs of monitors. This definition is used in the following context: \( \text{mon-pl} \ (\text{map} \ \alpha \ w) \) is the set of monitors used by a word \( w \) w.r.t. the abstraction \( \alpha \).

**definition**

\[ \text{mon-pl} w = \text{foldl} \ (\text{op} \ \cup) \ {} \ (\text{map} \ \lambda e. \text{fst} e \cup \text{snd} e) \ w \]

**lemma** \( \text{mon-pl-empty}[\text{simp}]: \text{mon-pl} \ [] = {} \)

---

\(^1\)Reentrant locks can always be simulated by non-reentrant ones, at the cost of a worst-case exponential blowup of the program size.
lemma \text{mon-pl-cons}[simp]: mon-pl (e\#w) = fst e \cup snd e \cup mon-pl w
by (unfold mon-pl-def) (simp, subst foldl-un-empty-eq, auto)

lemma \text{mon-pl-unconc}: \forall b. mon-pl (a@b) = mon-pl a \cup mon-pl b
by (induct a) auto

lemma \text{mon-pl-ileq}: w \preceq w' \implies mon-pl w \subseteq mon-pl w'
by (induct rule: less-eq-list-induct) auto

lemma \text{mon-pl-set}: mon-pl w = \bigcup \{ \text{fst e} \cup \text{snd e} | \text{e} \in \text{set w} \}
by (unfold mon-pl-def) (safe, auto simp add: Bex-def foldl-set)

fun \text{cil} :: \gamma a list \Rightarrow (\gamma a \Rightarrow (\gamma m \Rightarrow \gamma m \times \gamma m)) \Rightarrow \gamma a list set
\text{cil} :: 'a list \Rightarrow ('a \Rightarrow ('m set \times 'm set)) \Rightarrow 'a list \Rightarrow 'a list set
\text{cil} :: \gamma a list \Rightarrow (\gamma a \Rightarrow (\gamma m \Rightarrow \gamma m \times \gamma m)) \Rightarrow \gamma a list set

— Interleaving with the empty word results in the empty word
| w \otimes_\alpha [] = \{ w \}
| w \otimes_\alpha [] = \{ w \}

— If both words are not empty, we can take the first step of one word, interleave the rest with the other word and then append the first step to all result set elements, provided it does not allocate a monitor that is used by the other word
| e1\#w1 \otimes_\alpha e2\#w2 = (if fst (\alpha e1) \cap mon-pl (map \alpha (e2\#w2)) = \{ \} then e1\#(w1 \otimes_\alpha e2\#w2)
else \{ \}) \cup (if fst (\alpha e2) \cap mon-pl (map \alpha (e1\#w1)) = \{ \} then e2\#(e1\#w1 \otimes_\alpha w2)
else \{ \})

Note that this definition allows reentrant monitors, because it only checks that a monitor that is going to be entered by one word is not used in the other word. Thus the same word may enter the same monitor multiple times.

The next lemmas are some auxiliary lemmas to simplify the handling of the consistent interleaving operator.

lemma \text{cil-last-case-split}[cases set, case-names left right]:
| \in e1\#w1 \otimes_\alpha e2\#w2;
| \in w\#e1\#w1 \otimes_\alpha e2\#w2;
| \in w\#e1\#w1 \otimes_\alpha e2\#w2;
| \in w\#(w1 \otimes_\alpha e2\#w2);
| \in w\#(e1\#w1 \otimes_\alpha w2);
| \in w\#(e1\#w1 \otimes_\alpha w2);
| \in \text{fst (\alpha e1)} \cap mon-pl (map \alpha (e2\#w2)) = \{ \} \implies P;
| \in \text{fst (\alpha e2)} \cap mon-pl (map \alpha (e1\#w1)) = \{ \} \implies P
| \implies P
by (auto elim: list-set-cons-cases split: split-if-asm)

lemma \text{cil-cases}[cases set, case-names both-empty left-empty right-empty app-left app-right]:
proof (induct wa α wb rule: cil.induct)
  case 1 thus ⟨case by simp next
  case 2 thus ⟨case by simp next
  case (3 ea wa’ α eb wb’)
  from 3.prems(1) show ⟨thesis proof (cases rule: cil-last-case-split)
    case (left w) from 3.prems(5) ⟨OF left(1) - left(2,3)⟩ show ⟨thesis by simp
    next
    case ⟨right w) from 3.prems(6) ⟨OF right(1) - right(2,3)⟩ show ⟨thesis by simp
    qed
  qed

lemma cil-induct'[case-names both-empty left-empty right-empty append]: []
  ⟨α. P α [] []⟩;
  ⟨α ad ae. P α [] (ad # ae)⟩;
  ⟨α z aa. P α (z # aa) []⟩;
  ⟨α e1 w1 e2 w2. []
    ⟨fst (α e1) ∩ mon-pl (map α (e2 # w2)) = {}⟩ P α w1 (e2 # w2);
    ⟨fst (α e2) ∩ mon-pl (map α (e1 # w1)) = {}⟩ P α (e1 # w1) w2
  ⟩ P α (e1 # w1) (e2 # w2)
  ⟩ P α wa wb
  apply ⟨induct wa α wb rule: cil.induct⟩
  apply ⟨case-tac w⟩
  apply ⟨auto⟩
  done

lemma cil-induct-fixα: []
  P α [] []:
  ⟨α ad ae. P α [] (ad # ae)⟩;
  ⟨α z aa. P α (z # aa) []⟩;
  ⟨α e1 w1 e2 w2. []
    ⟨fst (α e2) ∩ mon-pl (map α (e1 # w1)) = {}⟩ ⟹ P α (e1 # w1) w2;
    ⟨fst (α e1) ∩ mon-pl (map α (e2 # w2)) = {}⟩ ⟹ P α w1 (e2 # w2)
  ⟩ P α (e1 # w1) (e2 # w2)
  ⟩ P α v w
  apply ⟨induct v α w rule: cil.induct⟩
  apply ⟨case-tac w⟩
  apply ⟨auto⟩
  done
lemma cil-induct-fixα [case-names both-empty left-empty right-empty append]:

\[
\begin{align*}
P & \alpha \sqsubseteq \emptyset; \\
\forall \alpha. P & \alpha \sqsubseteq \emptyset \alpha; \\
\forall \alpha. P & \alpha \emptyset \alpha; \\
fst(\alpha e1) \cap \text{mon-pl}(\text{map } \alpha (e2 \# w2)) = \{} \Rightarrow P \alpha w1 (e2 \# w2); \\
fst(\alpha e2) \cap \text{mon-pl}(\text{map } \alpha (e1 \# w1)) = \{} \Rightarrow P \alpha (e1 \# w1) w2 \\
\Rightarrow P \alpha (e1 \# w1) (e2 \# w2)
\end{align*}
\]

apply (induct wa α wb rule: cil.induct)
apply (case-tac w)
apply auto
done

lemma [simp]: \(w \otimes \alpha = \{w\}\)

by (cases w, auto)

lemma cil-contains-empty [rule-format, simp]: \(\emptyset \in \text{wa} \otimes \alpha \text{wb}\) = (\(\text{wa}=\emptyset \land \text{wb}=\emptyset\))

by (induct wa α wb rule: cil.induct) auto

lemma cil-cons-cases [cases set, case-names left right]: \(e \# w \in w1 \otimes \alpha w2; \)

\[
\begin{align*}
\forall \alpha e w1 w2. & \{e \# w \in w1 \otimes \alpha w2; \text{fst } \alpha e \cap \text{mon-pl}(\text{map } \alpha w2) = \{} \Rightarrow P; \\
\forall \alpha e w2 w1. & \{e \# w2 \in w1 \otimes \alpha w2; \text{fst } \alpha e \cap \text{mon-pl}(\text{map } \alpha w1) = \{} \Rightarrow P \\
\Rightarrow P w \alpha w1 w2
\end{align*}
\]

by (cases rule: cil-cases) auto

lemma cil-set-induct [induct set, case-names empty left right]: \(\forall \alpha w1 w2. \)

\[
\begin{align*}
\forall w1 \in w1 \otimes \alpha w2; \\
\forall w1. & w1 \emptyset w2; \\
\forall w1 \in w1 \otimes \alpha w2; \\
\forall w1 w2. & \{w1 \otimes \alpha w2; \text{fst } \alpha e \cap \text{mon-pl}(\text{map } \alpha w2) = \{} \Rightarrow P \alpha w1 w2; \\
\forall w1 w2. & \{w2 \emptyset w1; \text{fst } \alpha e \cap \text{mon-pl}(\text{map } \alpha w1) = \{} \Rightarrow P \alpha w1 w2 \Rightarrow P \alpha w1 w2
\end{align*}
\]

by (induct w) (auto intro!: cil-contains-empty elim: cil-cons-cases)

lemma cil-set-induct-fixα [induct set, case-names empty left right]: \(\forall w1 w2. \)

\[
\begin{align*}
\forall w1 w2. & \{w1 \emptyset w2; \\
\forall w1 \in w1 \otimes \alpha w2; \\
\forall e w1. & \{w1 \emptyset w2; \text{fst } \alpha e \cap \text{mon-pl}(\text{map } \alpha w2) = \{} \Rightarrow P \alpha w1 w2; \\
\forall e w1. & \{w1 \emptyset w2; \text{fst } \alpha e \cap \text{mon-pl}(\text{map } \alpha w1) = \{} \Rightarrow P \alpha w1 w2 \Rightarrow P \alpha w1 w2
\end{align*}
\]

by (induct w) (auto intro!: cil-contains-empty elim: cil-cons-cases)

lemma cil-cons1: \(\forall w1 w2. \text{fst } \alpha e \cap \text{mon-pl}(\text{map } \alpha w1 w2) = \{} \Rightarrow e \# w \in e \# w \otimes \alpha w1 w2\)

by (cases \(w1 w2\)) auto
2.2 Properties of consistent interleaving

— Consistent interleaving is a restriction of interleaving

**lemma cil-subset-il**: \( w \otimes \alpha w' \subseteq w \otimes w' \)

**apply** (induct \( w \alpha w' \) rule: cil.induct)

**apply** simp-all

**apply** safe

**apply** auto

**done**

**lemma cil-subset-il'**: \( w \in w_1 \otimes \alpha w_2 \Rightarrow w \in w_1 \otimes w_2 \)

**using** cil-subset-il by (auto)

— Consistent interleaving preserves the set of letters of both operands

**lemma cil-set**: \( w \in w_1 \otimes \alpha w_2 \Rightarrow \text{set } w = \text{set } w_1 + \text{set } w_2 \)

**by** (induct rule: cil-set-induct-fix \( \alpha \)) auto

— Consistent interleaving preserves the length of both operands

**lemma cil-length**: \( w \in w_1 \otimes \alpha w_2 \Rightarrow \text{length } w = \text{length } w_1 + \text{length } w_2 \)

**by** (induct rule: cil.induct) auto

— Consistent interleaving contains all letters of each operand in the original order

**lemma cil-ilq**: \( w \in w_1 \otimes \alpha w_2 \Rightarrow w_1 \sqsubseteq w \land w_2 \sqsubseteq w \)

**by** (intro conjI cil-subset-il' ilq-interleave)

— Consistent interleaving is commutative and associative

**lemma cil-commute**: \( w \otimes \alpha w' = w' \otimes \alpha w \)

**by** (induct rule: cil.induct) auto

**lemma cil-assoc1**: \![ \forall w_1 w_2 w_3. \[ w \in w_1 \otimes \alpha w_3; w \in w_1 \otimes w_2 \] \]

\[ \Rightarrow \exists w. w_1 \otimes \alpha w \land w \in w_2 \otimes \alpha w_3 \]

**proof** (induct \( w \) rule: length-compl-induct)

**case** Nil thus \( \forall \) case by auto

next

**case** (Cons \( e \) \( w \)) from Cons.prems(1) show \( \forall \) case proof (cases rule: cil-cons-cases)

**case** (left \( w' \)) with Cons.prems(2) have \( e \# w' \in w_1 \otimes \alpha w_2 \) by simp

thus \( \forall \) thesis proof (cases rule: cil-cons-cases[case-names left' right'])

**case** (left' \( w' \))

from Cons.hyps[OF - left(2) left'(2)] obtain \( w r \) where IHAPP: \( w \in w_1' \otimes \alpha w \land w \in w_2 \otimes \alpha w_3 \) by blast

have \( e \# w \in w_1' \otimes \alpha w \land w \) proof (rule cil-cons1[OF IHAPP(1)])

from left left' cil-mon-pl[OF IHAPP(2)] show \( \text{fst } (\alpha e) \cap \text{mon-pl } (\text{map } \alpha \)
\( \text{wr} \) = \{ \} by auto

qed

thus \( \text{thesis using IHAPP(2) left' by blast} \)

next

case (right' \( w3' \)) from Cons.hyps[OF - left(2) right'(2)] obtain \( \text{wr} \) where

IHAPP: \( w \in w1 \otimes_\alpha \text{wr} \) \( w1 \otimes_\alpha w3 \) by blast

from IHAPP(2) left have \( e\# w1 \otimes_\alpha w3 \) by (auto intro: cil-cons1)

moreover from right' IHAPP(1) have \( e\# w \in w1 \otimes_\alpha e\# \text{wr} \) by (auto intro: cil-cons2)

ultimately show \( \text{thesis using right' by blast} \)

qed

next

case (right \( w3' \)) from Cons.hyps[OF - right(2) Cons.prems(2)] obtain \( \text{wr} \) where

IHAPP: \( w \in w1 \otimes_\alpha \text{wr} \) \( w2 \otimes_\alpha w3' \) by blast

from IHAPP(2) right cil-mon-pl[OF Cons.prems(2)] have \( e\# w \in w2 \otimes_\alpha e\# \text{wr} \) by (auto intro: cil-cons2)

moreover from IHAPP(1) right cil-mon-pl[OF Cons.prems(2)] have \( e\# w \in w1 \otimes_\alpha e\# \text{wr} \) by (auto intro: cil-cons2)

ultimately show \( \text{thesis using right by blast} \)

qed

qed

lemma cil-assoc2:

assumes \( A: \text{wr} \in w1 \otimes_\alpha \text{wr} \) and \( B: \text{wr} \in w2 \otimes_\alpha w3 \)

shows \( \exists \text{wr}. w \in w1 \otimes\alpha w3 \wedge \text{wr} \in w1 \otimes_\alpha w2 \)

proof

from A have \( A': \text{wr} \in w1 \otimes_\alpha \text{wr} \) by (simp add: cil-commute)

from B have \( B': \text{wr} \in w2 \otimes_\alpha w3 \) by (simp add: cil-commute)

from cil-assoc1[OF \( A' B' \)] obtain \( \text{wr} \) where \( w \in w3 \otimes_\alpha \text{wr} \wedge \text{wr} \in w2 \otimes_\alpha w1 \)

by blast

thus \( \text{thesis by (auto simp add: cil-commute)} \)

qed

— Parts of the abstraction can be moved to the operands

lemma cil-map: \( w \in w1 \otimes_{(\alpha f)} w2 \Rightarrow \text{map} f w \in \text{map} f w1 \otimes_\alpha \text{map} f w2 \)

proof (induct rule: cil-set-induct-fix\( \alpha \))

case empty thus \( \text{case by auto} \)

next

case (left \( e \) \( w' \) \( w1' \) \( w2' \))

have \( f e \# \text{map} f w1' \otimes_\alpha \text{map} f w2 \) proof (rule cil-cons1)

from left(2) have \( \text{fst} ((\alpha f) e) \cap \text{mon-pl} (\text{map} \alpha (\text{map} f w2)) = \{ \} \) by (simp only: map-map[symmetric])

thus \( \text{fst} ((\alpha f) e) \cap \text{mon-pl} (\text{map} \alpha (\text{map} f w2)) = \{ \} \) by (simp only: o-apply)

qed (rule left(3))

thus \( \text{case by simp} \)

next

case (right \( e \) \( w' \) \( w2' \) \( w1' \))
have f e ≠ map f w' ∈ map f w1 ⊗_α f e ≠ map f w2' proof (rule cil-cons2)
  from right(2) have fst ((α◦f)e) ∩ mon-pl (map α (map f w1)) = {} by (simp only: map-map [symmetric])
    thus fst (α (f e)) ∩ mon-pl (map α (map f w1)) = {} by (simp only: o-apply)
  qed (rule right(3))
thus ?case by simp
qed

end

3 Acquisition Histories

theory AcquisitionHistory
imports ConsInterleave
begin

The concept of acquisition histories was introduced by Kahlon, Ivancic, and Gupta [5] as a bounded size abstraction of executions that acquire and release locks that contains enough information to decide consistent interleavability. In this work, we use this concept for reentrant monitors. As in Section 2, we encode monitor usage information in pairs of sets of monitors, and regard lists of such pairs as (abstract) executions. An item (E, U) of such a list describes a sequence of steps of the concrete execution that first enters the monitors in E and then passes through the monitors in U. The monitors in E are never left by the execution. Note that due to the syntactic binding of monitors to the program structure, any execution of a single thread can be abstracted to a sequence of (E, U)-pairs. Restricting the possible schedules (see Section 8) will allow us to also abstract executions reaching a single program point to a sequence of such pairs.

We want to decide whether two executions are interleavable. The key observation of [5] is, that two executions e and e' are not interleavable if and only if there is a conflicting pair (m, m') of monitors, such that e enters (and never leaves) m and then uses m' and e' enters (and never leaves) m' and then uses m.

An acquisition history is a map from monitors to set of monitors. The acquisition history of an execution maps a monitor m that is allocated at the end of the execution to all monitors that are used after or in the same step that finally enters m. Monitors that are not allocated at the end of an execution are mapped to the empty set. Though originally used for a setting without reentrant monitors, acquisition histories also work for our setting with reentrant monitors.

This theory contains the definition of acquisition histories and acquisition history interleavability, an ordering on acquisition histories that reflects the
blocking potential of acquisition histories, and a mapping function from paths to acquisition histories that is shown to be compatible with monitor consistent interleaving.

3.1 Definitions

Acquisition histories are modeled as functions from monitors to sets of monitors. Intuitively \( m' \in h \) models that an execution finally is in \( m \), and monitor \( m' \) has been used (i.e. passed or entered) after or at the same time \( m \) has been finally entered. By convention, we have \( m \in h \) or \( h = \{ \} \).

**Definition**

\[
ab == \{ (h::'m \Rightarrow 'm set) . \forall m. h m = \{ \} \lor m \in h m \}
\]

**Lemma**

**ah-cases[cases set]:** \([ h \in \ab; h m = \{ \} \implies P ; m \in h m \implies P ] \implies P \)

**by** (unfold \( ah \)-def) blast

3.2 Interleavability

Two acquisition histories \( h_1 \) and \( h_2 \) are considered interleavable, iff there is no conflicting pair of monitors \( m_1 \) and \( m_2 \), where a pair of monitors \( m_1 \) and \( m_2 \) is called conflicting iff \( m_1 \) is used in \( h_2 \) after entering \( m_2 \) and, vice versa, \( m_2 \) is used in \( h_1 \) after entering \( m_1 \).

**Definition**

\[
ah-il :: ('m \Rightarrow 'm set) \Rightarrow ('m \Rightarrow 'm set) \Rightarrow bool \ (infix \ ['\ast'] \ 65)
\]

**where**

\[
h_1 \ ['\ast'] \ h_2 == \neg(\exists m_1 m_2. m_1 \in h_2 m_2 \land m_2 \in h_1 m_1)
\]

From our convention, it follows (as expected) that the sets of entered monitors (lock-sets) of two interleavable acquisition histories are disjoint

**Lemma**

**ah-il-lockset-disjoint:**

\[
[h_1 \in \ab; h_2 \in \ab; h_1 \ ['\ast'] h_2] \implies h_1 m = \{ \} \lor h_2 m = \{ \}
\]

**by** (unfold \( ah-il \)-def) (auto elim: ah-cases)

Of course, acquisition history interleavability is commutative

**Lemma**

**ah-il-commute:** \( h_1 \ ['\ast'] h_2 \implies h_2 \ ['\ast'] h_1 \)

**by** (unfold \( ah-il \)-def) auto

3.3 Used monitors

Let’s define the monitors of an acquisition history, as all monitors that occur in the acquisition history

**Definition**

\[
mon-ah :: ('m \Rightarrow 'm set) \Rightarrow 'm set
\]

**where**

\[
mon-ah h == \bigcup \{ h(m) \mid m. True \}
\]
3.4 Ordering

The element-wise subset-ordering on acquisition histories intuitively reflects the blocking potential: The bigger the acquisition history, the fewer acquisition histories are interleavable with it.

Note that the Isabelle standard library automatically lifts the subset ordering to functions, so we need no explicit definition here.

— The ordering is compatible with interleavability, i.e. smaller acquisition histories are more likely to be interleavable.

\[ \text{lemma } ah-leq-il: \ h1 [\ast] h2; h1' \leq h1; h2' \leq h2 \implies h1' [\ast] h2' \]

\[ \text{by (unfold ah-il-def le-fun-def [where } b='a set]) blast+ } \]

\[ \text{lemma } ah-leq-il-left: \ h1 [\ast] h2; h1' \leq h1 \implies h1' [\ast] h2 \text{ and } ah-leq-il-right: \ h1 [\ast] h2; h2' \leq h2 \implies h1 [\ast] h2' \]

\[ \text{by (unfold ah-il-def le-fun-def [where } b='a set]) blast+ } \]

3.5 Acquisition histories of executions

Next we define a function that abstracts from executions (lists of enter/use pairs) to acquisition histories

\[ \text{primrec } \alpha ah :: ('m set \times 'm set) list \Rightarrow 'm \Rightarrow 'm set \text{ where } \]

\[ \alpha ah [] m = \{\} \]

\[ \alpha ah (e#w) m = (if m\in fst e \text{ then } fst e \cup snd e \cup mon-pl w \text{ else } \alpha ah w m) \]

— \( \alpha ah \) generates valid acquisition histories

\[ \text{lemma } ah-ah: \alpha ah w \in ah \]

\[ \text{apply (induct w)} \]

\[ \text{apply (unfold ah-def)} \]

\[ \text{apply simp} \]

\[ \text{apply (fastforce split: split-if-asm)} \]

\[ \text{done} \]

\[ \text{lemma } ah-hd: \ [m\in fst e; x\in fst e \cup snd e \cup mon-pl w] \implies x\in \alpha ah (e\#w) m \]

\[ \text{by auto} \]

\[ \text{lemma } ah-tl: \ [m\notin fst e; x\in ah w m] \implies x\in ah (e\#w) m \]

\[ \text{by auto} \]

\[ \text{lemma } ah-cases[cases set, case-names hd tl]: [ \]

\[ x\in ah w m; \]

\[ !!e w'. [w=e\#w'; m\in fst e; x\in fst e \cup snd e \cup mon-pl w'] \implies P; \]

\[ !!e w'. [w=e\#w'; m\notin fst e; x\in ah w' m] \implies P \]

\[ ] \implies P \]

\[ \text{by (cases w) (simp-all split: split-if-asm)} \]

\[ \text{lemma } ah-cons-cases[cases set, case-names hd tl]: [ \]

\[ x\in ah (e\#w') m; \]

\[ [m\in fst e; x\in fst e \cup snd e \cup mon-pl w'] \implies P; \]

\[ [m\notin fst e; x\in ah w' m] \implies P \]
\[ \iff P \]
by (simp-all split: split-if-asm)

**Lemma** mon-ah-subset: mon-ah (\(\alpha w\)) \(\subseteq\) mon-pl \(w\)
by (induct \(w\)) (auto simp add: mon-ah-def)

--- Subwords generate smaller acquisition histories

**Lemma** \(\alpha\)-ileq: \(w_1 \leq w_2 \implies \alpha w_1 \leq \alpha w_2\)
**Proof** (induct rule: less-eq-list-induct)

**Case** empty **thus** ?case by (unfold le-fun-def [where \(b=\alpha set\)], simp)

**Next**

**Case** (\(\text{drop } l' l a\)) show ?case
**Proof** (unfold le-fun-def [where \(b=\alpha set\], intro allI subsetI)

**Fix** \(m x\)
**Assume** \(A: x \in \alpha ah l' m\)
**With** \(\text{drop(2) have } x \in \alpha ah l m \text{ by (unfold le-fun-def [where \(b=\alpha set\], auto})
**Moreover hence** \(x \in \text{mon-pl } l\) using mon-ah-subset[unfolded mon-ah-def] by fast
**Ultimately show** \(x \in \alpha ah (a \# l) m\) by auto
**Qed**

**Next**

**Case** (\(\text{take } a b l' l\)) show ?case
**Proof** (unfold le-fun-def [where \(b=\alpha set\], intro allI subsetI)

**Fix** \(m x\)
**Assume** \(A: x \in \alpha ah (a \# l') m\)
**Thus** \(x \in \alpha ah (b \# l) m\)
**Proof** (cases rule: \(\alpha ah\)-cons-cases)

**Case** \(\text{hd}\)
**With** \(\text{mon-pl-ileq[OF take.hyps(2)] and } \langle a = b\rangle\)
**Show** ?thesis by auto

**Next**

**Case** \(\text{tl}\)
**With** take.hyps(3)[unfolded le-fun-def [where \(b=\alpha set\]] and \(\langle a = b\rangle\)
**Show** ?thesis by auto
**Qed**
**Qed**

We can now prove the relation of monitor consistent interleavability and interleavability of the acquisition histories.

**Lemma** ah-interleavable1:
\(w \in w_1 \otimes_{\alpha} w_2 \implies \alpha ah (\text{map } \alpha w_1) [*] \alpha ah (\text{map } \alpha w_2)\)
--- The lemma is shown by induction on the structure of the monitor consistent interleaving operator

**Proof** (induct \(w \alpha w_1 w_2\) rule: cil-set-induct-fix\(\alpha\))

**Case** empty **show** ?case by (simp add: ah-il-def) --- The base case is trivial by the definition of \(\text{op} [*]\)

**Next**
--- Case: First step comes from the left word
\textbf{case (left \( e \, w' \, w_1' \, w_2 \)) show ?case}
\begin{proof} (rule \textit{cccontr}) \hspace{1em} \text{We do a proof by contradiction}
\hspace{2em} \text{Assume there is a conflicting pair in the acquisition histories}
\hspace{3em} \text{\textbf{assume} \( \neg \, \alpha \, a \, h \, (\text{map} \, \alpha \, (\, e \, \# \, w_1')) \, [\ast] \, \neg \, \alpha \, a \, h \, (\text{map} \, \alpha \, w_2) \)}
\hspace{4em} \text{\textbf{then obtain} \( \forall \, m_1 \, m_2 \) where \( \text{CPAIR} \): \( m_1 \in \alpha \, a \, h \, (\text{map} \, \alpha \, (\, e \, \# \, w_1')) \) \( m_2 \in \alpha \, a \, h \, (\text{map} \, \alpha \, w_2) \) \textbf{by (unfold ah-il-def, blast)}
\hspace{5em} \text{It comes either from the first step or not}
\hspace{6em} \text{\textbf{from} \( \text{CPAIR}(1) \) \textbf{have} \( m_2 \in \text{fst} \, (\alpha \, e) \land m_1 \in \text{fst} \, (\alpha \, e) \cup \text{snd} \, (\alpha \, e) \cup \text{mon-pl} \, (\text{map} \, \alpha \, w_1') \) \( \cup \) \( m_2 \in \text{fst} \, (\alpha \, e) \land m_1 \in \alpha \, a \, h \, (\text{map} \, \alpha \, w_1') \) \( m_2) \) \textbf{(is ?CASE1 \lor ?CASE2)}
\hspace{7em} \textbf{by (auto split: split-if-asm)}
\hspace{8em} \textbf{moreover \{}
\hspace{9em} \text{\textbf{assume} \( \text{?CASE1} \) \textbf{hence} \( C: \, m_2 \in \text{fst} \, (\alpha \, e) \) ..}
\hspace{10em} \text{Because the paths are consistently interleavable, the monitors entered in the first step must not occur in the other path}
\hspace{11em} \text{\textbf{from} \( \text{left}(2) \) \textbf{mon-ah-subset[of map} \, \alpha \, w_2] \textbf{have} \, \text{fst} \, (\alpha \, e) \land \text{mon-ah} \, (\alpha \, a \, h \, (\text{map} \, \alpha \, w_2)) \) \textbf{=} \{ \} \textbf{by auto}
\hspace{12em} \text{But this is a contradiction to being a conflicting pair}
\hspace{13em} \textbf{with} \( C \) \( \text{CPAIR}(2) \) \textbf{have} \( \text{False} \) \textbf{by (unfold mon-ah-def, blast)}
\hspace{14em} \textbf{moreover \{}
\hspace{15em} \text{\textbf{assume} \( \text{?CASE2} \) \textbf{hence} \( C: \, m_1 \in \alpha \, a \, h \, (\text{map} \, \alpha \, w_1') \) \( m_2) \) ..}
\hspace{16em} \text{But this is a contradiction to the induction hypothesis, that says that the acquisition histories of the tail of the left path and the right path are interleavable}
\hspace{17em} \textbf{with} \( C \) \( \text{CPAIR}(2) \) \textbf{have} \( \text{False} \) \textbf{by (unfold ah-il-def, blast)}
\hspace{18em} \textbf{\} ultimately show} \( \text{False} \) ..
\hspace{19em} \textbf{qed}
\hspace{2em} \textbf{next}
\hspace{3em} \text{\textbf{\textbf{case (right} \, e \, w' \, w_2' \, w_1)} show} \ \textbf{?case}
\hspace{4em} \textbf{proof (rule \textit{cccontr})}
\hspace{5em} \text{\textbf{assume} \( \neg \, \alpha \, a \, h \, (\text{map} \, \alpha \, w_1') \, [\ast] \, \neg \, \alpha \, a \, h \, (\text{map} \, \alpha \, (\, e \, \# \, w_2')) \)}
\hspace{6em} \textbf{then obtain} \( \forall \, m_1 \, m_2 \) where \( \text{CPAIR} \): \( m_1 \in \alpha \, a \, h \, (\text{map} \, \alpha \, w_1') \) \( m_2 \in \alpha \, a \, h \, (\text{map} \, \alpha \, (\, e \, \# \, w_2')) \) \textbf{by (unfold ah-il-def, blast)}
\hspace{7em} \textbf{from} \( \text{CPAIR}(2) \) \textbf{have} \( m_1 \in \text{fst} \, (\alpha \, e) \land m_2 \in \text{fst} \, (\alpha \, e) \cup \text{snd} \, (\alpha \, e) \cup \text{mon-pl} \, (\text{map} \, \alpha \, w_2) \) \( \cup \) \( m_1 \in \text{fst} \, (\alpha \, e) \land m_2 \in \alpha \, a \, h \, (\text{map} \, \alpha \, w_2) \) \textbf{by (auto split: split-if-asm)}
\hspace{8em} \textbf{moreover \{}
\hspace{9em} \text{\textbf{assume} \( \text{?CASE1} \) \textbf{hence} \( C: \, m_1 \in \text{fst} \, (\alpha \, e) \) ..}
\hspace{10em} \textbf{from} \( \text{right}(2) \) \textbf{mon-ah-subset[of map} \, \alpha \, w_1] \textbf{have} \, \text{fst} \, (\alpha \, e) \land \text{mon-ah} \, (\alpha \, a \, h \, (\text{map} \, \alpha \, w_1)) \) \textbf{=} \{ \} \textbf{by auto}
\hspace{11em} \textbf{with} \( C \) \( \text{CPAIR}(1) \) \textbf{have} \( \text{False} \) \textbf{by (unfold mon-ah-def, blast)}
\hspace{12em} \textbf{moreover \{}
\hspace{13em} \text{\textbf{assume} \( \text{?CASE2} \) \textbf{hence} \( C: \, m_2 \in \alpha \, a \, h \, (\text{map} \, \alpha \, w_2) \) \( m_1) \) ..
\hspace{14em} \textbf{qed}
with right(3) CPAIR(1) have False by (unfold ah-il-def, blast)
} ultimately show False ..
qed

lemma ah-interleavable2:
assumes A: αah (map α w1) [*] αah (map α w2)
shows w1 ⊗_α w2 ≠ {} 
— This lemma is shown by induction on the sum of the word lengths
proof —
— To apply this induction in Isabelle, we have to rewrite the lemma a bit
{ fix n
have !!w1 w2. [[αah (map α w1) [*] αah (map α w2); n=length w1 + length w2]] ⟹ w1 ⊗_α w2 ≠ {} 
proof (induct n rule: nat-less-induct [case-names I])
— We first rule out the cases that one of the words is empty
  case (I n w1 w2) show ?case proof (cases w1)
  — If the first word is empty, the lemma is trivial
  case Nil with I.prems show ?thesis by simp
next
  case (Cons e1 w1’) note CONS1 = this show ?thesis proof (cases w2)
  — If the second word is empty, the lemma is also trivial
  case Nil with I.prems show ?thesis by simp
next
— The interesting case is if both words are not empty
  case (Cons e2 w2’) note CONS2 = this
  — In this case, we check whether the first step of one of the words can
  — The first step of the first word can safely be executed
  — From the induction hypothesis, we get that there is a consistent
  — And because the first step of the first word can be safely executed, we
  can prepend it to that consistent interleaving
  have w1’ ⊗_α w2 ≠ {} proof —
  from I.prems(1) CONS1 ah-leq-il-left[OF - αah-ileq[OF le-list-map, OF less-eq-list-drop[OF order-refl]]] have αah (map α w1’) [*] αah (map α w2)
by fast
moreover from CONS1 I.prems(2) have length w1’+length w2 < n
by simp
ultimately show ?thesis using I.hyps by blast
qed
— And because the first step of the first word can be safely executed, we
can prepend it to that consistent interleaving
with cil-consI[OF - True] CONS1 show ?thesis by blast
next
  case False note C1 = this
  show ?thesis proof (cases fst (α e2) ∩ mon-pl (map α w1) = {}) 
  case True — The first step of the second word can safely be executed
  — This case is shown analogously to the latter one
have $w_1 \otimes w_2' \neq \{\} 
\text{proof} \quad 
\text{from } I \text{.prems}(1) \text{ CONS2 ah-leq-il-right}[OF - \alpha \text{ah-ileq}[OF \text{le-list-map, } 
\text{OF less-eq-list-drop}[OF \text{order-refl}]]] \text{ have } \alpha \text{ah (map } \alpha \text{ w1) [s]} \alpha \text{ah (map } \alpha \text{ w2')} 
\text{by fast} 
\text{moreover from } \text{CONS2 I \text{.prems}(2) have } \text{length w1+length w2'} < n \text{ by simp} 
\text{ultimately show } ?\text{thesis using } I \text{.hyps by blast} 
\text{next} 
\text{case } \text{False note } C2 = \text{this} \quad \text{— Neither first step can safely be executed.} 
\text{This is exactly the situation from that we can extract a conflicting pair} 
\text{from } C1 \text{ C2 obtain } m_1 m_2 \text{ where } m_1 \in \text{fst (} \alpha \text{ e1) m1} \in \text{mon-pl (map } \alpha \text{ w2) m2} \in \text{fst (} \alpha \text{ e2) m2} \in \text{mon-pl (map } \alpha \text{ w1) by blast} 
\text{with } \text{CONS1 CONS2 have } m_2 \in \alpha \text{ah (map } \alpha \text{ w1) m1 m1} \in \alpha \text{ah (map } \alpha \text{ w2) m2} \text{ by auto} 
\text{— But by assumption, there are no conflicting pairs, thus we get a contradiction} 
\text{with } I \text{.prems(1) have } \text{False by (unfold ah-il-def) blast} 
\text{thus } ?\text{thesis ..} 
\text{qed} 
\text{qed} 
\text{qed} 
\text{qed} 
\text{qed} 
\text{)} \text{with } A \text{ show } ?\text{thesis by blast} 
\text{qed} 

Finally, we can state the relationship between monitor consistent interleaving and interleaving of acquisition histories

\text{theorem } ah\text{-interleavable:} 
\quad (\alpha \text{ah (map } \alpha \text{ w1) [s]} \alpha \text{ah (map } \alpha \text{ w2)}) \iff (w1 \otimes w2' \neq \{\}) 
\text{using } ah\text{-interleavable1 ah\text{-interleavable2 by blast} 

3.6 Acquisition history backward update

We define a function to update an acquisition history backwards. This function is useful for constructing acquisition histories in backward constraint systems.

\text{definition} 
\quad ah\text{-update} :: (\text{'m} \Rightarrow \text{'m set}) \Rightarrow (\text{'m set} \ast \text{'m set}) \Rightarrow \text{'m set} \Rightarrow (\text{'m} \Rightarrow \text{'m set}) 
\text{where} 
\quad ah\text{-update} h F M m == \text{if } m \in \text{fst F then } \text{fst F} \cup \text{snd F} \cup M \text{ else } h \text{ m} 

Intuitively, \text{ah-update} h (E, U) M m means to prepend a step (E, U) to the acquisition history h of a path that uses monitors M. Note that we need the extra parameter M, since an acquisition history does not contain information.
about the monitors that are used on a path before the first monitor that will not be left has been entered.

**Lemma ah-update-cons:** \( \alpha (e\#w) = \text{ah-update} (\alpha ah) e (\text{mon-pl} w) \)
by (auto intro: ext simp add: ah-update-def)

The backward-update function is monotonic in the first and third argument as well as in the used monitors of the second argument. Note that it is, in general, not monotonic in the entered monitors of the second argument.

**Lemma ah-update-mono:** \( h \leq h'; F=F'; M \subseteq M' \) 
by (auto simp add: ah-update-def le-fun-def)

**Lemma ah-update-mono2:** \( h \leq h'; U \subseteq U'; M \subseteq M' \) 
by (auto simp add: ah-update-def le-fun-def)

---

## 4 Labeled transition systems

**Theory LTS**

**Imports** Main

**Begin**

Labeled transition systems (LTS) provide a model of a state transition system with named transitions.

### 4.1 Definitions

An LTS is modeled as a ternary relation between start configuration, transition label and end configuration.

**Type-synonym** \( ('c,'a) \text{LTS} = ('c \times 'a \times 'c) \text{set} \)

Transitive reflexive closure

**Inductive-set**

\( \text{trcl} :: ('c,'a) \text{LTS} \Rightarrow ('c,'a \text{ list}) \text{LTS} \)
for \( t \)
where
\[ \begin{align*}
\text{empty[simp]}: & (c,[]) \in \text{trcl} t \\
| \text{cons[simp]}: & [(c,a,c')] \in t; (c',w,c'') \in \text{trcl} t \implies (c,a\#w,c'') \in \text{trcl} t
\end{align*} \]

### 4.2 Basic properties of transitive reflexive closure

**Lemma trcl-empty-cons:** \( (c,[]; c') \in \text{trcl} t \implies (c=c') \)
by (auto elim: trcl.cases)

**Lemma trcl-empty-simp[simp]:** \( (c,[]; c') \in \text{trcl} t = (c=c') \)
by (auto elim: trcl.cases intro: trcl.intros)

---

18
lemma trcl-single[simp]: \((c,[a],c') \in trcl t\) = \((c,a,c') \in t\)
by (auto elim: trcl_cases)

lemma trcl-uncons: \( (c,a#w,c') \in trcl t \implies \exists \, ch \cdot (c,a,ch) \in t \land (ch,w,c') \in trcl t \)
by (auto elim: trcl_cases)

lemma trcl-uncons-cases: \[
\begin{array}{l}
  (c,a#w,c') \in trcl S; \\
  \Rightarrow \exists! \, ch. \, [(c,a,ch) \in S; (ch,w,c') \in trcl S] \implies P \\
\end{array}
\]
by (blast dest: trcl-uncons)

lemma trcl-one-elem: \( (c,e,c') \in t \implies (c,[e],c') \in trcl t \)
by auto

lemma trcl-unconsE[cases set, case-names split]: \[
\begin{array}{l}
  (c,e\#w,c') \in trcl S; \\
  \Rightarrow \forall \, ch. \, [(c,e,ch) \in S; (ch,w,c') \in trcl S] \implies P \\
\end{array}
\]
by (blast dest: trcl-uncons)

lemma trcl-unconsE[cases set, case-names split]: \[
\begin{array}{l}
  ((s,c),e\#w,(s',c')) \in trcl S; \\
  \Rightarrow \forall \, sh \, ch. \, [[(s,c),e,(sh,ch)] \in S; ((sh,ch),w,(s',c')) \in trcl S] \implies P \\
\end{array}
\]
by (fast dest: trcl-uncons)

lemma trcl-concat: \( \exists! \, c \cdot [(c,w1,c') \in trcl t \land (c',w2,c'') \in trcl t] \implies (c,w1@w2,c'') \in trcl t \)

proof (induct w1)
  case Nil thus \( ?case \) by (subgoal-tac c=c') auto
next
  case (Cons a w) thus \( ?case \) by (auto dest: trcl-uncons)
qed

lemma trcl-concat: \( \exists! \, c \cdot (c,w1@w2,c') \in trcl t \)
\( \Rightarrow \exists \, ch \cdot (c,w1,ch) \in trcl t \land (ch,w2,c') \in trcl t \)

proof (induct w1)
  case Nil hence \( (c,[],c) \in trcl t \land (c,w2,c') \in trcl t \) by auto
  thus \( ?case \) by fast
next
  case (Cons a w1) note IHP = this
  hence \( (c,a\#(w1@w2),c') \in trcl t \) by simp
  with trcl-uncons obtain chh where \( (c,a,chh) \in t \land (chh,w1@w2,c') \in trcl t \) by fast
  moreover with IHP obtain ch where \( (chh,w1,ch) \in trcl t \land (ch,w2,c') \in trcl t \) by fast
  ultimately have \( (c,a\#w1, ch) \in trcl t \land (ch,w2,c') \in trcl t \) by auto
  thus \( ?case \) by fast
qed
4.2.1  Appending of elements to paths

lemma trcl-cons2: \[(c,w,ch)\in trcl T; (ch,e,c')\in T \implies (c,w@[e],c')\in trcl T\]
by (auto dest: trcl-concat iff add: trcl-single)

lemma trcl-uncons: \[(e,w@[e],c')\in trcl T \implies \exists ch. (c,w,ch)\in trcl T \land (ch,e,c')\in T\]
by (force dest: trcl-unconcate)

lemma trcl-rev-induct [induct set, consumes 1, case-names empty snoc]: !! c'. [\[(c,w,c')\in trcl S; \]
\[!!c. P c [] c; \]
\[!!c w c' e c''. [\[(c,w,c')\in trcl S; (c',e,c'')\in S; P c w c' \implies P e (w@[e]) c''\]
]\]\[\implies P c w c'\]
by (induct w rule: rev-induct) (auto dest: trcl-rev-unscons)

lemma trcl-rev-conses: !! c'. [\[(c,w,e,c')\in trcl S; \]
\[[w=[]]; c=e''\implies P; \]
\[!!ch e wh. \[w=wh@[e]; (c,wh,ch)\in trcl S; (ch,e,c')\in S \implies P\]
]\]\[\implies P\]
by (induct w rule: rev-induct) (simp, blast dest: trcl-rev-unscons)

lemma trcl-cons2: \[(c,e,ch)\in T; (ch,f,c')\in T \implies (c,[e,f],c')\in trcl T\]
by auto

4.2.2  Transitivity reasoning setup

declare trcl-cons2 [trans] — It’s important that this is declared before trcl-concat, because we want trcl-concat to be tried first by the transitivity reasoner

declare cons [trans]

declare trcl-concat [trans]

declare trcl-rev-cons [trans]

4.2.3  Monotonicity

lemma trcl-mono: !! A B. A \subseteq B \implies trcl A \subseteq trcl B
apply (clarsimp)
apply (erule trcl.induct)
apply auto
done

lemma trcl-inter-mono: \[x\in trcl (S\cap R) \implies x\in trcl S\]
proof
  assume \[x\in trcl (S\cap R)\]
  with trcl-mono[of \(S\cap R\) \(S\)] show \[x\in trcl S\] by auto
next
  assume \[x\in trcl (S\cap R)\]
  with trcl-mono[of \(S\cap R\) \(R\)] show \[x\in trcl R\] by auto
qed
4.2.4 Special lemmas for reasoning about states that are pairs

lemmas trcl-pair-induct = trcl.induct[of (xa1,xa2) xb (xa1,xa2), split-format (complete), consumes 1, case-names empty cons]
lemmas trcl-rev-pair-induct = trcl-rev-induct[of (xa1,xa2) xb (xa1,xa2), split-format (complete), consumes 1, case-names empty snoc]

4.2.5 Invariants

lemma trcl-prop-trans[cases set, consumes 1, case-names empty steps]: []
(c,w,c′)∈trcl S;
[c=c′; w=[]] ⊢ P;
[c∈Domain S; c′∈Range (Range S)] ⊢ P
⊢ P
apply (erule-tac trcl-rev-cases)
apply auto
apply (erule trcl.cases)
apply auto
done

end

5 Thread Tracking

theory ThreadTracking
imports Main ~~/src/HOL/Library/Multiset LTS Misc
begin

This theory defines some general notion of an interleaving semantics. It
defines how to extend a semantics specified on a single thread and a context
to a semantic on multisets of threads. The context is needed in order to
keep track of synchronization.

5.1 Semantic on multiset configuration

The interleaving semantics is defined on a multiset of stacks. The thread to
make the next step is nondeterministically chosen from all threads ready to
make steps.

definition gtr gtrs == { (##s##+c,e,##s′##+c′) | s c e s′ c′. ((s,c),e,(s′,c′))∈gtrs }

lemma gtrl-s: ((s,c),e,(s′,c′))∈gtrs ⊢ (##s##+c,e,##s′##+c′)∈gtr gtrs
by (unfold gtr-def, auto)

lemma gtrl: ((s,c),w,(s′,c′))∈trcl gtrs
⇒ (##s##+c,w,##s′##+c′)∈trcl (gtr gtrs)
by (induct rule: trcl-pair-induct) (auto dest: gtrl-s)
lemma gtrE: [\]
  (c,e,c')\in gtr T;
  \forall s. c s' c'. [ c = \#s# + ce; c' = \#s'# + ce'; ((s, ce), e, (s', ce')) \in T ] \implies P
]\implies P
by (unfold gtr-def) blast

lemma gtr-empty-conf-s[simp]:
  (\#), w, c')\in gtr S
(c, w, \#)\in gtr S
by (auto elim: gtrE)

lemma gtr-empty-conf1[simp]: ((\#), w, c')\in trcl (gtr S) \iff (w=\[] \land c' = \#)
by (induct w) (auto dest: trcl-uncons)

lemma gtr-empty-conf2[simp]: ((c, w, \#))\in trcl (gtr S) \iff (w=\[] \land c = \#)
by (induct w rule: rev-induct) (auto dest: trcl-rev-uncons)

lemma gtr-find-thread: [\]
  (c, e, c')\in gtr gtrs;
  \forall s. c s' c'. [ c = \#s# + ce; c' = \#s'# + ce'; ((s, ce), e, (s', ce')) \in gtrs ] \implies P
]\implies P
by (unfold gtr-def) auto

lemma gtr-step-cases[cases set, case-names loc other]: [\]
  (\#\#), w, e, c')\in gtr gtrs;
  \forall s'. c'. [ c' = \#s'# + ce'; ((s, ce), e, (s', ce')) \in gtrs ] \implies P;
  \forall c. c ce c'. [ c = \#s# + ce; c' = \#s'# + ce'; ((s, \#s#) + ce), e, (s', ce')) \in gtrs ] \implies P
]\implies P
by (auto elim!: gtr-find-thread mset-single-cases)

lemma gtr-rev-cases[cases set, case-names loc other]: [\]
  (\#\#), c, e, c')\in gtr gtrs;
  \forall s. c. [ c = \#s# + ce; ((s, ce), e, (s', ce')) \in gtrs ] \implies P;
  \forall c. c ce c'. [ c = \#s# + ce; c' = \#s'# + ce; ((ss, ce), e, (ss', \#s'#) + ce)) \in gtrs ] \implies P
]\implies P
by (auto elim!: gtr-find-thread mset-single-cases)

5.2 Invariants

lemma gtr-preserve-s: [\]
  (c, e, c')\in gtr T;
  P c;
  \forall s. c s' c'. [ P (\#s#) + ce; ((s, ce), e, (s', ce')) \in T ] \implies P (\#s'#) + c'
]\implies P c'
by (unfold gtr-def) blast

lemma gtr-preserve: [\]
  (c, w, c')\in trcl (gtr T);
P c;

22
\[ s \in s' \in c. [\{s\} + c]; (s, c, e, (s', c')) \in T ] \implies P \{s'\} + c' \]

\[ \implies P c' \]

apply (induct rule: trcl.induct)
apply simp
apply (subgoal-tac P c')
apply blast
apply (blast intro: gtr-preserve-s)
done

5.3 Context preservation assumption

We now assume that the original semantics does not modify threads in the context, i.e. it may only add new threads to the context and use the context to obtain monitor information, but not change any existing thread in the context. This assumption is valid for our semantics, where the context is just needed to determine the set of allocated monitors. It allows us to generally derive some further properties of such semantics.

locale env-no-step =
  fixes gtrs :: ((s × s multiset), l) LTS
  assumes env-no-step-s[cases set, case-names csp]:
    \[ ((s, c), e, (s', c')) \in gtrs; csp. c' = csp + c \implies P \] \implies P

— The property of not changing existing threads in the context transfers to paths

lemma (in env-no-step) env-no-step[cases set, case-names csp]:
  \[ ((s, c), w, (s', c')) \in trcl gtrs;
     csp. c' = csp + c \implies P \]
\implies P

proof –
  have \((s, c), w, (s', c')\) \in trcl gtrs \implies \exists csp. c' = csp + c
    proof (induct rule: trcl-pair-induct)
    case empty thus \?case by (auto intro: exI[of - {#}])
  next
    case (cons s c e sh ch w s' c') note IHP=this
    from env-no-step-s[OF IHP(1)] obtain csp where ch=csp+ch by auto
    moreover from IHP(3) obtain csp' where c'=csp'+c by auto
    ultimately have c'=csp'+csph+c by (simp add: union-assoc)
    thus \?case by blast
  qed

  moreover assume \((s, c), w, (s', c')\) \in trcl gtrs \implies csp. c' = csp + c \implies P
  ultimately show \?thesis by blast
  qed

The following lemma can be used to make a case distinction how a step operated on a given thread in the end configuration:

loc The thread made the step

spawn The thread was spawned by the step

env The thread was not involved in the step

23
lemma (in env-no-step) rev-cases-p[cases set, case-names loc spawn env]:

assumes STEP: \( (c,e,\{s'\#\}+ce) \in \text{gtrs} \) and
LOC: \( !s \text{ ce. } \) \( c=\{s\#\}+ce; ((s,ce),e,(s',ce')) \in \text{gtrs} \) \( \Rightarrow P \) and
SPAWN: \( !s s' ce \text{ csp.} \)
\[
\begin{align*}
  & (c=\{s\#\}+ce; ce'=\{s'\#\}+csp+ce; \\
  & ((ss,ce),e,(ss',\{s'\#\}+csp+ce)) \in \text{gtrs} \\
  \Rightarrow & P \text{ and} \\
  & ENV: !s s' ce \text{ csp.} \\
  & (c=\{s\#\}+\{s'\#\}+ce+cc \in \text{gtrs by simp-all} \\
  & \text{from env-no-step-s[OF CASE'(3)] obtain} \text{ csp where} \text{ EQ: } \{s'\#\}+cc = csp + ce \text{ by blast} \\
  \text{thus } \text{?thesis proof (cases rule: mset-unplus-dist-cases)} \\
  \text{case left note } CC=\text{this} \\
  \text{with } CASE'= c = \{s\#\} + cc ce' = \{s'\#\} + cc ((ss, ce), e, ss', \{s'\#\} + cc) \in \text{gtrs by simp-all} \\
  \text{moreover from } CC(2) \text{ have } \{s'\#\}+cc = \{s\#\} + (csp-\{s'\#\}) + ce \text{ by (auto simp add: union-assoc)} \\
  \text{by (simp add: union-assoc) ultimately show } \text{?thesis using } CASE'(1,3) \text{ CASE(2) SPAWN by auto} \\
  \text{next} \\
  \text{case right note } CC=\text{this} \\
  \text{from } CC(1) \text{ CASE'(1) have } c=\{s\#\}+\{s'\#\} + (ce - \{s'\#\}) \text{ by (simp add: union-ac)} \\
  \text{moreover from } CC(2) \text{ CASE'(2) have } ce'=\{s\#\}+csp+(ce-\{s'\#\}) \text{ by (simp add: union-ac)} \\
  \text{moreover from } CC(2) \text{ have } \{s'\#\} + cc = csp + (\{s'\#\} + (ce-\{s'\#\})) \text{ by (simp add: union-ac) ultimately show } \text{?thesis using } CASE'(3) \text{ CASE(3) CC(1) ENV by auto} \\
  \text{qed}
\end{align*}
\]

5.4 Explicit local context

In the multiset semantics, a single thread has no identity. This may become a problem when reasoning about a fixed thread during an execution. For example, in our constraint-system-based approach the operational characterization of the least solution of the constraint system requires to state properties of the steps of the initial thread in some execution. With the multiset semantics, we are unable to identify those steps among all steps. There are many solutions to this problem, for example, using thread ids...
either as part of the thread’s configuration or as part of the whole configuration by using lists of stacks or maps from ids to stacks as configuration datatype.

In the following we present a special solution that is strong enough to suit our purposes but not meant as a general solution.

Instead of identifying every single thread uniquely, we only distinguish one thread as the local thread. The other threads are environment threads. We then attach to every step the information whether it was on the local or on some environment thread.

We call this semantics loc/env-semantics in contrast to the multiset-semantics of the last section.

5.4.1 Lifted step datatype

datatype 'a el-step = LOC 'a | ENV 'a

definition
  loc w == filter (λe. case e of LOC a ⇒ True | ENV a ⇒ False) w

definition
  env w == filter (λe. case e of LOC a ⇒ False | ENV a ⇒ True) w

definition
  le-rem-s e == case e of LOC a ⇒ a | ENV a ⇒ a

Standard simplification lemmas

lemma loc-env-simps[simp]:
  loc [] = []
  env [] = []
by (unfold loc-def env-def) auto

lemma loc-single[simp]: loc [a] = (case a of LOC e ⇒ [a] | ENV e ⇒ [])
by (unfold loc-def) (auto split: el-step.split)
lemma loc-uncons[simp]:
  loc (a#b) = (case a of LOC e ⇒ [a] | ENV e ⇒ [] ) @ loc b
by (unfold loc-def) (auto split: el-step.split)
lemma loc-unconc[simp]: loc (a@b) = loc a @ loc b
by (unfold loc-def, simp)

lemma env-single[simp]: env [a] = (case a of LOC e ⇒ [] | ENV e ⇒ [a])
by (unfold env-def) (auto split: el-step.split)
lemma env-uncons[simp]:
  env (a#b) = (case a of LOC e ⇒ [] | ENV e ⇒ [a]) @ env b
by (unfold env-def) (auto split: el-step.split)
lemma env-unconc[simp]: env (a@b) = env a @ env b
by (unfold env-def, simp)
The following simplification lemmas are for converting between paths of the multiset- and loc/env-semantics

**lemma le-rem-simps [simp]:**
- `le-rem-s (LOC a) = a`
- `le-rem-s (ENV a) = a`
  by (unfold le-rem-s-def, auto)

**lemma le-rem-id-simps [simp]:**
- `le-rem-s ◦ LOC = id`
- `le-rem-s ◦ ENV = id`
  by (auto intro: ext)

**lemma le-rem-id-map [simp]:**
- `map le-rem-s (map LOC w) = w`
- `map le-rem-s (map ENV w) = w`
  by auto

**lemma env-map-env [simp]:**
- `env (map ENV w) = map ENV w`
  by (unfold env-def)

**lemma env-map-loc [simp]:**
- `env (map LOC w) = []`
  by (unfold env-def)

**lemma loc-map-env [simp]:**
- `loc (map ENV w) = []`
  by (unfold loc-def)

**lemma loc-map-loc [simp]:**
- `loc (map LOC w) = map LOC w`
  by (unfold loc-def)

5.4.2 Definition of the loc/env-semantics

type-synonym 's el-conf = ('s × 's multiset)

inductive-set
- `gtrp` :: ('s el-conf,'l LTS ⇒ ('s el-conf,'l el-step) LTS
  for S
  where
  `gtrp-loc`: `((s,c),e,(s',c'))∈S ⇒ ((s,c),LOC e,(s',c'))∈gtrp S`
  | `gtrp-env`: `((s,{#s#}+c),e,(s',{#s'}#)+c')∈S` ⇒ `((s',{#s'}#)+c'),ENV e,({#s'}#)+c')∈gtrp S`

5.4.3 Relation between multiset- and loc/env-semantics

**lemma gtrp2gtr-s:**
- `((s,c),e,(s',c'))∈gtrp T ⇒ (({#s#}+c),le-rem-s e,{{#s'}#}+c')∈gtr T`

**proof (cases rule: gtrp.cases, auto intro: gtrI-s)**
- `fix c c' e ss ss' assume ((ss,{#ss#}+c),e,(ss',{#ss'}#)+c')∈T`
  | `hence (({#ss#}+({#ss#}+c),e,{{#ss'}#}+(({#ss#}+c')∈gtr T by (auto intro: gtrI-s)``
  | `thus (({#ss#}+({#ss#}+c),e,{{#ss'}#}+c')∈gtr T by (auto simp add: union-ac)qed`
lemma \texttt{gtr2gtr}:
\[(s,c),w,(s',c')\in\text{trcl} (\text{gtr } T)\]
\[\implies ((\#s\#)+c,\text{map le-rem-s } w,(\#s'\#)+c')\in\text{trcl} (\text{gtr } T)\]
\[\text{by (induct rule: trcl-pair-induct) (auto dest: gtr2gtr)}\]

lemma (in \texttt{env-no-step}) \texttt{gtr2gtr-s} [cases set, case-names gtrp]:
\[\text{assumes } A: ((\#s\#)+c,e,c')\in\text{gtr gtrs}\]
\[\text{and CASE: } !s' ce' ee. \{e'=\{\#s'\#\}+ce'; e=\text{le-rem-s } ee; ((s,c),ee,(s',ce'))\in\text{gtrp gtrs}\]
\[\implies P\]
\[\text{shows } P\]
\[\text{using } A\]
\[\text{proof (cases rule: gtr-step-cases) }\]
\[\text{case (loc } s' \text{ ce') hence ((s,c),LOC e,(s',ce'))\in gtrp gtrs by (blast intro: gtrp-loc)}\]
\[\text{with loc(1) show } \text{thesis by (rule-tac CASE) auto}\]
\[\text{next}\]
\[\text{case (other cc ss ss' cc') from env-no-step-s[OF other(3)] obtain csp where}\]
\[\text{CE'FMT: } ce'=csp + ((\#s\#)+cc) .\]
\[\text{with other(3) have ((ss,\{\#s\#\}+cc),e,(ss',\{\#s\#\}+(csp+cc)))\in gtr by (auto simp add: union-ac)}\]
\[\text{from gtrp-env[OF this] other(1) have ((s, c), ENV e, s, \{\#s'\#\} + (csp + cc))}\]
\[\in gtrp gtrs by simp}\]
\[\text{moreover from other CE'FMT have } c' = \{\#s'\#\} + (\{\#s'\#\} + (csp + cc))\]
\[\text{by (simp add: union-ac)}\]
\[\text{ultimately show } \text{thesis by (rule-tac CASE) auto}\]
\[\text{qed}\]

lemma (in \texttt{env-no-step}) \texttt{gtr2gtr}[cases set, case-names gtrp]:
\[\text{assumes } A: ((\#s\#)+c,w,c')\in\text{trcl} (\text{gtr gtrs})\]
\[\text{and CASE: } !s' ce' ww. \{e'=\{\#s'\#\}+ce'; w=\text{map le-rem-s } ww; ((s,c),ww,(s',ce'))\in\text{trcl} (\text{gtrp gtrs})\]
\[\implies P\]
\[\text{shows } P\]
\[\text{proof –}\]
\[\text{have } !s \quad (\{\#s\#\}+c,w,c')\in\text{trcl} (\text{gtr gtrs}) \implies \exists s' ce' \quad w=\text{map le-rem-s } ww \quad (s,c),ww,(s',ce'))\in\text{trcl} (\text{gtrp gtrs}) \text{ proof (induct w) }\]
\[\text{case Nil thus } \text{thesis by auto}\]
\[\text{next}\]
\[\text{case (Cons } e \quad w) \text{ then obtain ch where } \text{SPLIT: } (\{\#s\#\}+c,e,ch)\in\text{gtr gtrs} (ch,w,c')\in\text{trcl} (\text{gtr gtrs}) \text{ by (fast dest: trcl-unecons)}\]
\[\text{from gtr2gtr-s[OF SPLIT(1)] obtain sh ceh ee where } \text{FS: } ch = \{\#sh\#\} + ceh e = \text{le-rem-s } ee ((s, c), ee, sh, ceh) \in\text{gtrp gtrs by blast}\]
\[\text{moreover from } \text{FS}(1) \text{ SPLIT(2) Cons.hyps obtain } s' ce' \quad ww \text{ where } \text{IH: } c' = \{\#s'\#\}+ce' \quad w=\text{map le-rem-s } ww ((sh,ceh),ww,(s',ce'))\in\text{trcl} (\text{gtrp gtrs}) \text{ by blast}\]
\[\text{ultimately have } ((s,c),ee\#ww,(s',ce'))\in\text{trcl} (\text{gtrp gtrs}) \quad e\#w = \text{map le-rem-s } (ee\#ww) \text{ by auto}\]
\[\text{with } \text{IH(1) show } \text{thesis by } \text{iprover}\]
\[\text{qed}\]
with A CASE show ?thesis by blast
qed

5.4.4 Invariants

lemma gtrp-preserve-s:
assumes A: ((s,c),e,(s',c'))∈gtrp T
and INIT: P ((#s#)+c)
and PRES: !s c s' c' e. [P ((#s#)+c); ((s,c),e,(s',c'))∈T]
shows P ((#s'#)+c')
proof –
from gtr-preserve-s[OF gtrp2gtr-s[OF A], where P=P, OF INIT] PRES show P ((#s'#)+ c') by blast
qed

lemma gtrp-preserve:
assumes A: ((s,c),w,(s',c'))∈trcl (gtrp T)
and INIT: P ((#s#)+c)
and PRES: !s c s' c' e. [P ((#s#)+c); ((s,c),e,(s',c'))∈T]
shows P ((#s'#)+c')
proof –
from gtr-preserve[OF gtrp2gtr[OF A], where P=P, OF INIT] PRES show P ((#s'#)+ c') by blast
qed

end

6 Flowgraphs

theory Flowgraph
imports Main Misc
begin

We use a flowgraph-based program model that extends the one we used previously [6]. A program is represented as an edge annotated graph and a set of procedures. The nodes of the graph are partitioned by the procedures, i.e. every node belongs to exactly one procedure. There are no edges between nodes of different procedures. Every procedure has a distinguished entry and return node and a set of monitors it synchronizes on. Additionally, the program has a distinguished main procedure. The edges are annotated with statements. A statement is either a base statement, a procedure call or a thread creation (spawn). Procedure calls and thread creations refer to the called procedure or to the initial procedure of the spawned thread, respectively.
We require that the main procedure and any initial procedure of a spawned thread does not to synchronize on any monitors. This avoids that spawning of a procedure together with entering a monitor is available in our model as an atomic step, which would be an unrealistic assumption for practical problems. Technically, our model would become strictly more powerful without this assumption.

If we allowed this, our model would become strictly more powerful,

### 6.1 Definitions

**datatype** 
\[
(p, ba) \text{edgeAnnot} = \text{Base} \cdot ba \mid \text{Call} \cdot p \mid \text{Spawn} \cdot p
\]

**type-synonym** 
\[
(n, p, ba) \text{edge} = (n \times (p, ba) \text{edgeAnnot} \times n)
\]

**record** 
\[
(n, p, ba, m) \text{flowgraph-rec} = 
\]

**definition** 
\[
\text{initialproc} \ fg \ p = \ p = \text{main} \ fg \lor (\exists u. v. (u, \text{Spawn} p, v) \in \text{edges} \ fg)
\]

**lemma** 
\[
\text{main-is-initial}[\text{simp}]: \text{initialproc} \ fg \ (\text{main} \ fg)
\]

**locale** 
\[
\text{flowgraph} = 
\]

---

### 6.2 Basic properties

**lemma** 
\[
\text{spawn-no-mon}[\text{simp}]: (u, \text{Spawn} p, v) \in \text{edges} \ fg \implies \text{mon} \ fg \ p = \{}
\]

**using** 
\[
\text{initial-no-mon} \ \text{by (unfold initialproc-def, blast)}
\]

**lemma** 
\[
\text{main-no-mon}[\text{simp}]: \text{mon} \ (\text{main} \ fg) = \{}
\]

**using** 
\[
\text{initial-no-mon} \ \text{by (unfold initialproc-def, blast)}
\]

**lemma** 
\[
\text{entry-return-same-proc}[\text{simp}]:
\]
entry fg p = return fg p' \Rightarrow p=p'
apply (subgoal-tac proc-of fg (entry fg p) = proc-of fg (return fg p'))
apply (simp (no-asym-use))
by simp

lemma (in flowgraph) entry-entry-same-proc[simp]:
entry fg p = entry fg p' \Rightarrow p=p'
apply (subgoal-tac proc-of fg (entry fg p) = proc-of fg (entry fg p'))
apply (simp (no-asym-use))
by simp

lemma (in flowgraph) return-return-same-proc[simp]:
return fg p = return fg p' \Rightarrow p=p'
apply (subgoal-tac proc-of fg (return fg p) = proc-of fg (return fg p'))
apply (simp (no-asym-use))
by simp

6.3 Extra assumptions for flowgraphs

In order to simplify the definition of our restricted schedules (cf. Section 8), we make some extra constraints on flowgraphs. Note that these are no real restrictions, as we can always rewrite flowgraphs to match these constraints, preserving the set of conflicts. We leave it to future work to consider such a rewriting formally.

The background of this restrictions is that we want to start an execution of a thread with a procedure call that never returns. This will allow easier technical treatment in Section 8. Here we enforce this semantic restrictions by syntactic properties of the flowgraph.

The return node of a procedure is called isolated, if it has no incoming edges and is different from the entry node. A procedure with an isolated return node will never return. See Section 8.1 for a proof of this.

definition
isolated-ret fg p ==
(\forall u l. \neg(u,l,\text{return }fg\ p)\in\text{edges }fg) \land \text{entry }fg\ p \neq \text{return }fg\ p

The following syntactic restrictions guarantee that each thread’s execution starts with a non-returning call. See Section 8.1 for a proof of this.

locale eflowgraph = flowgraph +
— Initial procedure’s entry node isn’t equal to its return node
assumes initial-no-ret: initialproc fg p \implies \text{entry }fg\ p \neq \text{return }fg\ p
— The only outgoing edges of initial procedures’ entry nodes are call edges to procedures with isolated return node
assumes initial-call-no-ret: [initialproc fg p; (entry fg p,l,v)\in\text{edges }fg] \implies \exists p'. l=\text{Call }p' \land \text{isolated-ret }fg\ p'}
6.4 Example Flowgraph

This section contains a check that there exists a (non-trivial) flowgraph, i.e. that the assumptions made in the flowgraph and eflowgraph locales are consistent and have at least one non-trivial model.

```markdown
definition example-fg == ()
  edges = {((0::nat,0::nat), Call 1,(0,1)),
            ((1,0), Spawn 0, (1,0)),
            ((1,0), Call 0, (1,0))},
  main = 0,
  entry = λp. (p,0),
  return = λp. (p,1),
  mon = λp. if p=1 then {0} else {},
  proc-of = λ (p,x). p []

lemma exists-eflowgraph: eflowgraph example-fg
  apply (unfold-locales)
  apply (unfold example-fg-def)
  apply simp
  apply fast
  apply simp
  apply simp
  apply (simp add: initialproc-def)
  apply (simp add: initialproc-def)
  apply (simp add: initialproc-def isolated-ret-def)
  done
```

end

7 Operational Semantics

theory Semantics
imports Main Flowgraph ~~/src/HOL/Library/Multiset LTS Interleave ThreadTracking
begin

7.1 Configurations and labels

The state of a single thread is described by a stack of control nodes. The top node is the current control node and the nodes deeper in the stack are stored return addresses. The configuration of a whole program is described by a multiset of stacks.

Note that we model stacks as lists here, the first element being the top element.

type-synonym 'n conf = ('n list) multiset

A step is labeled according to the executed edge. Additionally, we introduce
a label for a procedure return step, that has no corresponding edge.

datatype ('p, 'ba) label = LBase 'ba | LCall 'p | LRet | LSpawn 'p

7.2 Monitors

The following defines the monitors of nodes, stacks, configurations, step labels and paths (sequences of step labels)

definition — The monitors of a node are the monitors the procedure of the node synchronizes on
mon-n fg n == mon fg (proc-of fg n)

definition — The monitors of a stack are the monitors of all its nodes
mon-s fg s == \bigcup \{ mon-n fg n \mid n . n \in set s \}

definition — The monitors of a configuration are the monitors of all its stacks
mon-c fg c == \bigcup \{ mon-s fg s \mid s . s :# c \}

— The monitors of a step label are the monitors of procedures that are called by this step

definition mon-e :: ('b, 'c, 'd, 'a, 'e) flowgraph-rec-scheme => (c', 'f) label => 'a set where
mon-e fg e = (case e of (LCall p) => mon fg p | _ => {})

lemma mon-e-simps [simp]:
  mon-e fg (LBase a) = {}
  mon-e fg (LCall p) = mon fg p
  mon-e fg (LRet) = {}
  mon-e fg (LSpawn p) = {}
  by (simp-all add: mon-e-def)

— The monitors of a path are the monitors of all procedures that are called on the path

definition mon-w fg w == \bigcup \{ mon-e fg e \mid e . e \in set w \}

lemma mon-s-alt: mon-s fg s == \bigcup (mon fg ' proc-of fg ' set s)
  by (unfold mon-s-def mon-n-def) (auto intro!: eq-reflection)
lemma mon-c-alt: mon-c fg c == \bigcup (mon-s fg ' set-mset c)
  by (unfold mon-c-def set-mset-def) (auto intro!: eq-reflection)
lemma mon-w-alt: mon-w fg w == \bigcup (mon-e fg ' set w)
  by (unfold mon-w-def) (auto intro!: eq-reflection)

lemma mon-sI: \[ n \in set s . m \in mon-n fg n \] => m \in mon-s fg s
  by (unfold mon-s-def, auto)
lemma mon-sD: m \in mon-s fg s => ? n \in set s . m \in mon-n fg n
by (unfold mon-s-def, auto)

lemma mon-n-same-proc:
  proc-of fg n = proc-of fg n' \implies mon-n fg n = mon-n fg n'
by (unfold mon-n-def, simp)

lemma mon-s-same-proc:
  proc-of fg ' set s = proc-of fg ' set s' \implies mon-s fg s = mon-s fg s'
by (unfold mon-s-alt, simp)

lemma (in flowgraph) mon-of-entry[simp]: mon-n fg (entry fg p) = mon fg p
by (unfold mon-n-def, simp add: entry-valid)

lemma (in flowgraph) mon-of-ret[simp]: mon-n fg (return fg p) = mon fg p
by (unfold mon-n-def, simp add: return-valid)

lemma mon-c-single[simp]: mon-c fg {#s#} = mon-s fg s
by (unfold mon-c-def) auto

lemma mon-s-single[simp]: mon-s fg [n] = mon-n fg n
by (unfold mon-s-def) auto

lemma mon-s-empty[simp]: mon-s fg [] = { }
by (unfold mon-s-def) auto

lemma mon-c-empty[simp]: mon-c fg {#} = { }
by (unfold mon-c-def) auto

lemma mon-s-unconc: mon-s fg (a@b) = mon-s fg a \cup mon-s fg b
by (unfold mon-s-def) auto

lemma mon-s-uncons[simp]: mon-s fg (a#as) = mon-n fg a \cup mon-s fg as
by (rule mon-s-unconc[where a=\{a\}, simplified])

lemma mon-c-unconc: mon-c fg (a+b) = mon-c fg a \cup mon-c fg b
by (unfold mon-c-def) auto

lemma mon-cI: \[ s:#c; m \in mon-s fg s] \implies m \in mon-c fg c
by (unfold mon-c-def, auto)

lemma mon-cD: \[ m \in mon-c fg c \] \implies \exists s. s:#c \land m \in mon-s fg s
by (unfold mon-c-def, auto)

lemma mon-s-mono: set s \subseteq set s' \implies mon-s fg s \subseteq mon-s fg s'
by (unfold mon-s-def) auto

lemma mon-c-mono: c \leq #c' \implies mon-c fg c \subseteq mon-c fg c'
by (unfold mon-c-def) (auto intro: mset-le-trans-elem)

lemma mon-w-empty[simp]: mon-w fg [] = { }
by (unfold mon-w-def, auto)

lemma mon-w-single[simp]: mon-w fg [e] = mon-e fg e
by (unfold mon-w-def, auto)

lemma mon-w-unconc: mon-w fg (wa@wb) = mon-w fg wa \cup mon-w fg wb
by (unfold mon-w-def) auto

lemma mon-w-uncons[simp]: mon-w fg (c#w) = mon-e fg e \cup mon-w fg w
by (rule mon-w-unconc[where wa=\{e\}, simplified])
**7.3 Valid configurations**

We call a configuration *valid* if each monitor is owned by at most one thread.

**definition**

\[
\text{valid } fg \, c \equiv \forall s \, s'. \{ \# s \# \} + \{ \# s' \# \} \leq c \Rightarrow \text{mon-s } fg \, s \cap \text{mon-s } fg \, s' = \{ \}
\]

**lemma valid-empty** [simp, intro!]: valid \( fg \, \{\#\} \)

by (unfold valid-def, auto)

**lemma valid-single** [simp, intro!]: valid \( fg \, \{\# s\#\} \)

by (unfold valid-def subset-mset-def) auto

**lemma valid-split1:**

valid \( fg \, (c + c') \) \( \Rightarrow \) valid \( fg \, c \land valid \, fg \, c' \land mon-c \, fg \, c \cap mon-c \, fg \, c' = \{ \} \)

apply (unfold valid-def)

apply (auto simp add: mset-le-incr-right)

apply (drule mon-cD)+

apply auto

apply (subgoal-tac \{\#s\#\} + \{\#sa\#\} \leq \# c+c')


done

**lemma valid-split2:**

\[
\begin{align*}
\text{valid } fg \, (c + c') & \Rightarrow \text{valid } fg \, (c + c') \\
\text{valid } fg \, c \land \text{valid } fg \, c' & \land \text{mon-c } fg \, c \cap \text{mon-c } fg \, c' = \{ \}
\end{align*}
\]

apply (unfold valid-def)

apply (intro impI allI)

apply (erule mset-2dist2-cases)

apply simp-all

apply (blast intro: mon-cI)+

done

**lemma valid-unconc:**

valid \( fg \, (c + c') \) \( \iff \) valid \( fg \, c \land valid \, fg \, c' \land mon-c \, fg \, c \cap mon-c \, fg \, c' = \{ \} \)

by (blast dest: valid-split1 valid-split2)

**lemma valid-no-mon:** mon-c \( fg \, c = \{ \} \) \( \Rightarrow \) valid \( fg \, c \)

**proof** (unfold valid-def, intro allI impI)

fix \( s \, s' \)

assume A: mon-c \( fg \, c = \{ \} \) and B: \( \{\#s\#\} + \{\#s'\#\} \leq \# \, c \)

from mon-c-mono[OF B, of fg] A have mon-s \( fg \, s = \{ \} \) mon-s \( fg \, s' = \{ \} \) by (auto simp add: mon-c-unconc)

thus mon-s \( fg \, s \cap mon-s \, fg \, s' = \{ \} \) by blast

qed
7.4 Configurations at control points

— A stack is at U if its top node is from the set U

\textbf{primrec} atU-s :: ‘n set ⇒ ‘n list ⇒ bool where
\begin{align*}
atU-s U [] &= False \\
atU-s U (u#r) &= (u \in U)
\end{align*}

\textbf{lemma} atU-s-decomp\texttt{[simp]}: atU-s U (s@s’) = (atU-s U s \lor (s=[] \land atU-s U s’))
\begin{enumerate}
\item by (induct s) auto
\end{enumerate}

— A configuration is at U if it contains a stack that is at U

\textbf{definition} atU U c == ∃s. s:#c \land atU-s U s

\textbf{lemma} atU-fmt: [atU U c; !!r. [ui#r :# c; ui\in U] \implies P] \implies P
\begin{enumerate}
\item apply (unfold atU-def)
\item apply auto
\item apply (case-tac s)
\item apply auto
\item done
\end{enumerate}

\textbf{lemma} atU-union-cases\texttt{[case-names left right, consumes 1]}:
\begin{enumerate}
\item atU U (c1+c2);
\item atU U c1 \implies P;
\item atU U c2 \implies P
\item \implies P
\end{enumerate}
\begin{enumerate}
\item by (unfold atU-def) (blast elim: mset-un-cases)
\end{enumerate}

\textbf{lemma} atU-add: atU U c \implies atU U (c+ce) \land atU U (ce+c)
\begin{enumerate}
\item by (unfold atU-def) (auto simp add: union-ac)
\end{enumerate}

\textbf{lemma} atU-union\texttt{[simp]}: atU U (c1+c2) = (atU U c1 \lor atU U c2)
\begin{enumerate}
\item by (auto simp add: atU-union-cases)
\end{enumerate}

\textbf{lemma} atU-empty\texttt{[simp]}: \neg atU U {#}
\begin{enumerate}
\item by (unfold atU-def, auto)
\end{enumerate}

\textbf{lemma} atU-single\texttt{[simp]}: atU U {#s#} = atU-s U s
\begin{enumerate}
\item by (unfold atU-def, auto)
\end{enumerate}

\textbf{lemma} atU-single-top\texttt{[simp]}: atU U {#u#r#} = (u\in U)
\begin{enumerate}
\item by (auto)
\end{enumerate}

\textbf{lemma} atU-exchange-stack: atU U ({#u#r#}+c) \implies atU U ({#u#r’#}+c)
\begin{enumerate}
\item by (simp)
\end{enumerate}

— A configuration is simultaneously at U and V if it contains a stack at U and another one at V

\textbf{definition} atUV U V c == ∃su sv. {#su#}+{#sv#} \leq# c \land atU-s U su \land atU-s V sv

\textbf{lemma} atUV-empty\texttt{[simp]}: \neg atUV U V {#}
by (unfold atUV-def) auto

**lemma** atUV-single[simp]: \(\neg atUV U V \{\#s\}\)

by (unfold atUV-def) auto

**lemma** atUV-union[simp]:
\[
atUV U V (c1 + c2) \leftrightarrow
( (atUV U V c1) \lor
( atUV U V c2) \lor
( atU U c1 \land atUV V c2) \lor
( atUV V c1 \land atU U c2)
)
\]
apply (unfold atUV-def atU-def)
apply (auto elim: mset-2dist2-cases intro: mset-le-incr-right iff add: mset-le-mono-add-single)
apply (subst union-commute)
apply (auto iff add: mset-le-mono-add-single)
done

**lemma** atUV-union-cases[case-names left right lr rl], consumes 1:

\[
[ [ atUV U V (c1 + c2); atUV U V c1 = \Rightarrow P; atUV U V c2 = \Rightarrow P; [atU U c1; atU V c2] = \Rightarrow P; [atUV V c1; atU U c2] = \Rightarrow P ] \Rightarrow P by auto
\]

7.5 Operational semantics

7.5.1 Semantic reference point

We now define our semantic reference point. We assess correctness and completeness of analyses relative to this reference point.

**inductive-set**

refpoint :: ('n,'p,'ba,'m,'more) flowgraph-rec-scheme \(\Rightarrow\)

('n conf \times (p,ba) label \times 'n conf) set

for \(fg\)

where

- A base edge transforms the top node of one stack and leaves the other stacks untouched.

  **refpoint-base**: \(\[ (u,Base a,v)\in edges fg; valid fg (\{\#u\#r\#\}+c) \]\)

  \(\Rightarrow (\{\#u\#r\#\}+c, LBase a,(\#v\#r\#\}+c)\in refpoint fg]\)

- A call edge transforms the top node of a stack and then pushes the entry node of the called procedure onto that stack. It can only be executed if all monitors the called procedure synchronizes on are available. Reentrant monitors are modeled here by checking availability of monitors just against the other stacks, not against the stack of the thread that executes the call. The other stacks are left untouched.

  **refpoint-call**: \(\[ (u,Call p,v)\in edges fg; valid fg (\{\#u\#r\#\}+c);\)
The interleaving semantics is generated using the general techniques from Section 5.

— A return step pops a return node from a stack. There is no corresponding flowgraph edge for a return step. The other stacks are left untouched.

refpoint-ret: \[ valid \ fg \ ((\#\ return \ fg \ p\#r\#)+c) \]  
\[ \{\{\#\ return \ fg \ p\#r\#\}+c\} \in \text{refpoint} \ fg \]

— A spawn edge transforms the top node of a stack and adds a new stack to the environment, with the entry node of the spawned procedure at the top and no stored return addresses. The other stacks are also left untouched.

refpoint-spawn: \[ ((u, \text{Spawn} \ p, v) \in \text{edges} \ fg; \ valid \ fg \ ((\#u\#r\#)+c)) \]  
\[ \{\{\#u\#r\#\}+c\}, \text{LSpawn} \ p, (\#v\#r\#)+\{\#|\text{entry} \ fg \ p\#|\}+c \} \in \text{refpoint} \ fg \]

Instead of working directly with the reference point semantics, we define the operational semantics of flowgraphs by describing how a single stack is transformed in a context of environment threads, and then use the theory developed in Section 5 to derive an interleaving semantics. Note that this semantics is also defined for invalid configurations (cf. Section 7.3). In Section 7.6.1 we will show that it preserves validity of a configuration, and in Section 7.6.2 we show that it is equivalent to the reference point semantics on valid configurations.

**inductive-set**

\[ \text{trss} :: (\text{'n'}, p, \text{'ba'}, \text{'m'}, \text{more}) \text{ flowgraph-rec-scheme} \Rightarrow \]
\[ ((\text{'n} \text{ list} \ast \text{'n} \text{ conf}) \ast (\text{'p}, \text{'ba} \text{ label} \ast (\text{'n} \text{ list} \ast \text{'n} \text{ conf}))) \text{ set} \]

for \( \text{fg} \)

where

\[ \text{trss-base: } [(u, \text{Base} \ a, v) \in \text{edges} \ fg] \Rightarrow \]
\[ ((u\#r, c), \text{LBase} a, (v\#r, c)) \in \text{trss} \ fg \]

\[ \text{trss-call: } [(u, \text{Call} \ p, v) \in \text{edges} \ fg; \ mon \ fg \ p \cap \text{mon-c} \ fg \ c = \{\}] \Rightarrow \]
\[ ((u\#r, c), \text{LCall} p, (\text{entry} \ fg \ p)\#v\#r, c)) \in \text{trss} \ fg \]

\[ \text{trss-ret: } (((\text{return} \ fg \ p)\#r), c), \text{LRet} (r, c)) \in \text{trss} \ fg \]

\[ \text{trss-spawn: } [(u, \text{Spawn} \ p, v) \in \text{edges} \ fg] \Rightarrow \]
\[ ((u\#r, c), \text{LSpawn} p, (v\#r, (\#|\text{entry} \ fg \ p|\#)+c)) \in \text{trss} \ fg \]

— The interleaving semantics is generated using the general techniques from Section 5

**abbreviation** \( \text{tr} \ where \ \text{tr} \ fg := \text{gtr} (\text{trss} \ fg) \)

— We also generate the loc/env-semantics

**abbreviation** \( \text{trp} \ where \ \text{trp} \ fg := \text{gtrp} (\text{trss} \ fg) \)

### 7.6 Basic properties

#### 7.6.1 Validity

**lemma** (in flowgraph) \( \text{trss-valid-preserve-s:} \]
\[ \{\text{valid} \ fg \ (\text{'s}s\#)+c\}; \{(s,c), e, (s',c')\} \in \text{trss} \ fg \] \[ \Rightarrow \] \[ \text{valid} \ fg \ (\text{'s}'s\#)+c' \]

**apply** (erule trss.cases)

**apply** (simp-all add: valid-unconc mon-c-unconc)
by (blast dest: mon-n same proc edges-part)+

**Lemma (in flowgraph) trss-valid-preserve:**
\[\{(s,c),w,(s',c')\} \in \text{trcl} (\text{trss} \ fg); \ valid \ fg (\{\#s\#\}+c) \Rightarrow \ valid \ fg (\{\#s'\#\}+c')\]
by (induct rule: trcl-pair-induct) (auto intro: trss-valid-preserve-s)

**Lemma (in flowgraph) tr-valid-preserve-s:**
\[\{(c,e,c')\} \in \text{tr} \ fg; \ valid \ fg \ c \Rightarrow \ valid \ fg \ c'\]
by (rule gtr-preserve-s[where \(P=\text{valid} \ fg\)]) (auto dest: trss-valid-preserve-s)

**Lemma (in flowgraph) tr-valid-preserve:**
\[\{(c,w,c')\} \in \text{trcl} (\text{tr} \ fg); \ valid \ fg \ c \Rightarrow \ valid \ fg \ c'\]
by (rule gtr-preserve[where \(P=\text{valid} \ fg\)]) (auto dest: trss-valid-preserve-s)

**Lemma (in flowgraph) trp-valid-preserve-s:**
\[\{(s,c),w,(s',c')\} \in \text{trp} \ fg; \ valid \ fg (\{\#s\#\}+c) \Rightarrow \ valid \ fg (\{\#s'\#\}+c')\]
by (rule gtrp-preserve-s[where \(P=\text{valid} \ fg\)]) (auto dest: trss-valid-preserve-s)

**Lemma (in flowgraph) trp-valid-preserve:**
\[\{(s,c),w,(s',c')\} \in \text{trcl} (\text{trp} \ fg); \ valid \ fg (\{\#s\#\}+c) \Rightarrow \ valid \ fg (\{\#s'\#\}+c')\]
by (rule gtrp-preserve[where \(P=\text{valid} \ fg\)]) (auto dest: trss-valid-preserve-s)

### 7.6.2 Equivalence to reference point

— The equivalence between the semantics that we derived using the techniques from Section 5 and the semantic reference point is shown nearly automatically.

**Lemma refpoint-eq-s:** \(\text{valid} \ fg \ c \Rightarrow \{(c,e,c')\} \in \text{refpoint} \ fg \leftrightarrow \{(c,e,c')\} \in \text{tr} \ fg\)

apply rule
apply (erule refpoint_cases)
apply (auto intro: gtrl-s trss.intros simp add: union_assoc)
apply (erule gtrE)
apply (erule trss_cases)
apply (auto intro: refpoint.intros simp add: union_assoc[symmetric])
done

**Lemma (in flowgraph) refpoint-eq:**
\(\text{valid} \ fg \ c \Rightarrow \{(c,w,c')\} \in \text{trcl} (\text{refpoint} \ fg) \leftrightarrow \{(c,w,c')\} \in \text{trcl} (\text{tr} \ fg)\)

**Proof**

have \(\{(c,w,c')\} \in \text{trcl} (\text{refpoint} \ fg) \Rightarrow \text{valid} \ fg \ c \Rightarrow \{(c,w,c')\} \in \text{trcl} (\text{tr} \ fg)\) by (induct rule: trcl.induct) (auto simp add: refpoint-eq-s trss-valid-preserve-s)
moreover have \(\{(c,w,c')\} \in \text{trcl} (\text{tr} \ fg) \Rightarrow \text{valid} \ fg \ c \Rightarrow \{(c,w,c')\} \in \text{trcl} (\text{refpoint} \ fg)\) by (induct rule: trcl.induct) (auto simp add: refpoint-eq-s trss-valid-preserve-s)
ultimately show \(\text{valid} \ fg \ c \Rightarrow \{(c,w,c')\} \in \text{trcl} (\text{refpoint} \ fg) = \{(c,w,c')\} \in \text{trcl} (\text{tr} \ fg)\) ..

qed

### 7.6.3 Case distinctions

**Lemma trss-c-cases-s [cases set, case-names no-spawn spawn]:**
\[\{(s,e),(s',e')\} \in \text{trss} \ fg;\]
\[
\begin{align*}
\text{lemma } & \text{trss-c-fmt-s: } \exists (s, c', c, (s', c')) \in \text{trss } fg \\
& \quad \Rightarrow \exists csp. \ c' = csp + c \land \hspace{1cm} (csp = \# \lor (\exists p. e = \text{LSpawn } p \land csp = \#(\text{entry } fg \ p \#))) \\
& \hspace{1cm} \text{by (force elim!): trss-c-cases-s}
\end{align*}
\]
trss.cases

hence !!s. s : # {entry fg p #} + csp \rightarrow \exists p u v. s = [entry fg p] \land (u, Spawn p, v) \in edges fg \land initialproc fg p using CSPFMT by (unfold initialproc-def, erule-tac mset-un-cases) (auto)

ultimately show ?case using IHP(3) by blast
qed
qed
qed

lemma (in flowgraph) c-of-initial-no-mon:
assumes A: !!s. s :# csp \rightarrow \exists p. s = [entry fg p] \land initialproc fg p
shows mon-c fg csp = {}
by (unfold mon-c-def) (auto dest: A initial-no-mon)

lemma (in flowgraph) trss-c-no-mon-s:
assumes A: \((s,c),w,(s',c')\)\in trss fg
shows mon-c fg c' = mon-c fg c
using A
proof (erule-tac trss-c-cases-s)
assume c' = c thus ?thesis by simp
next
fix p assume EFMT: c = LSpawn p and C'FMT: c' = # {entry fg p #} + c
from EFMT obtain u v where (u, Spawn p, v) \in edges fg using A by (auto elim: trss.cases)
with spawn-no-mon have mon-c fg # {entry fg p #} = {} by simp
with C'FMT show ?thesis by (simp add: mon-c-unconc)
qed

corollary (in flowgraph) trss-c-no-mon:
\((s,c),w,(s',c')\)\in trcl (trss fg) \Rightarrow mon-c fg c' = mon-c fg c
apply (auto elim!: trss.cases simp add: mon-c-anconc)
proof
fix csp x
assume x\in mon-c fg csp
then obtain s where s :# csp and M: x\in mon-s fg s by (unfold mon-c-def, auto)
moreover assume \forall s. 0 < count csp s \rightarrow (\exists p. s = [entry fg p] \land (\exists u v. (u, Spawn p, v) \in edges fg) \land initialproc fg p)
ultimately obtain p u v where s = [entry fg p] and (u, Spawn p, v) \in edges fg by blast
hence mon-s fg s = {} by (simp)
with M have False by simp
thus x\in mon-c fg c..
qed
lemma (in flowgraph) trss-spawn-no-mon-step[simp]:
\[(s, c), L\text{Spawn } p, (s', c')\in trss \text{fg} \implies mon \text{fg } p = \] 
by (auto elim: trss.cases)

lemma trss-no-empty-s[simp]: \[([], c, s', c')\in trss \text{fg} = False\]
by (auto elim!: trss.cases)

lemma trss-no-empty[simp]:

assumes \(A\): \(([], c, w, (s', c'))\in trcl \text{ (trss fg)}\)
shows \(w = [] \land s' = [] \land c = c'\)
proof –
moreover {
  fix \(s\)
  have \((s, c, w, (s', c'))\in trcl \text{ (trss fg)} \implies s = [] \implies w = [] \land s' = [] \land c = c'\)
  by (induct rule: trcl-pair-induct) auto
}
ultimately show \(?thesis by blast\)
qed

lemma trs-step-cases[cases set, case-names NO-SPAWN SPAWN]:

assumes \(A\): \((c, e, c')\in tr \text{fg} \)
assumes \(A\)-NO-SPAWN: \(!s ce s' esp. [\]
\(((s, ce), e, (s', ce))\in tr \text{fg}; c=\#s\# + ce; c'=\#s'\# + ce \]
\] \implies P\)
assumes \(A\)-SPAWN: \(!s ce s' p. [\]
\(((s, ce), L\text{Spawn } p, (s', \#[entry } fg \text{ p]\# + ce))\in tr \text{fg}; c=\#s\# + ce; c'=\#s'\# + \#[entry } fg \text{ p]\# + ce; e=L\text{Spawn } p\]
\] \implies P\)
shows \(P\)
proof –
from \(A\) show \(?thesis proof \ (erule-tac gtr-find-thread)\)
fix \(s ce s' ce'\)
assume FMT: \(c = \#s\# + ce c' = \#s'\# + ce'\)
assume B: \(((s, ce), e, s', ce')\in tr \text{fg thus } ?thesis proof \ (cases rule: trss-c-cases-s)\)
case no-spawn thus \(?thesis using FMT B by (\text{-}) \ (rule A-NO-SPAWN, auto)\)
next
case (spawn p) thus \(?thesis using FMT B by (\text{-}) \ (rule A-SPAWN, auto simp add: union-assoc)\)
qed

qed
7.7 Advanced properties

7.7.1 Stack composition / decomposition

**lemma** trss-stack-comp-s:
\[(s,c),e,(s',c')\in\text{trss }fg \implies ((s@r,c),e,(s'@r,c'))\in\text{trss }fg\]
by (auto elim!: trss.cases intro: trss.intros)

**lemma** trss-stack-comp:
\[(s,c),w,(s',c')\in\text{trcl }\text{(trss }fg) \implies ((s@r,c),w,(s'@r,c'))\in\text{trcl }\text{(trss }fg)\]
**proof** (induct rule: trcl-pair-induct)
- case empty thus \text{?case by auto}
- next
  - case (cons s c e sh ch w s' c') \hspace{1em} \text{note IHP=this}
  - from trss-stack-comp-s[OF IHP(1)] have ((s @ r, c), e, sh @ r, ch) \in trss fg .
  - also note IHP(3)
  - finally show ?case .
qed

**lemma** trss-stack-decomp-s:
\[
\exists sp'. s' = sp' @ r \land ((s,c),e,(sp',c'))\in\text{trss }fg
\]
by (cases s, simp) (auto intro: trss.intros elim!: trss.cases)

**lemma** trss-find-return:
\[
((s@r,c),w,(r,c'))\in\text{trcl }\text{(trss }fg); \quad \text{!!wa wb ch. } \begin{align*}
  w &= wa @ wb; ((s,c),wa,([],ch))\in\text{trcl }\text{(trss }fg); \\
  (r, ch), wb, (r, c')\in\text{trcl }\text{(trss }fg) \end{align*} \implies P
\]
- If \( s = [] \), the proposition follows trivially
- apply (cases \( s = [] \))
- apply fastforce
**proof**
- \text{— For } s \neq [()], we use induction by \( w \)
  - have \( \text{IM: !!s. } c. \begin{align*} ((s@r,c),w,(r,c'))\in\text{trcl }\text{(trss }fg); \quad s' \neq [()] \end{align*} \implies \exists wa wb ch. \begin{align*} w &= wa @ wb \land ((s,c),wa,([],ch))\in\text{trcl }\text{(trss }fg) \land (r, ch), wb, (r, c')\in\text{trcl }\text{(trss }fg) \end{align*} \)
**proof** (induct \( w \))
- case Nil thus \text{?case by (auto)}
- next
  - case (Cons e w) \hspace{1em} \text{note IHP=this}
    - then obtain sh ch where SPLIT1: \((s@r,c),e,(sh, ch))\in\text{trss }fg \text{ and SPLIT2:} \((sh, ch),w,(r,c'))\in\text{trcl }\text{(trss }fg)\] by (fast dest: trcl-uncons)
      { assume CASE: e=LRet
        \begin{itemize}
          \item with SPLIT1 obtain p where EDGE: s@=return fg p \# sh c=ch by (auto elim!: trss.cases)
        \end{itemize}
        \text{with IHP(3) obtain ss where SHFMT: } s = \text{return fg p} \neq ss \text{ sh}=ss@r\]
        \text{by (cases s, auto)}
      } assume CC: ss[]
      \begin{itemize}
        \item with SHFMT have \( \exists ss. ss[] \land sh=ss@r \) by blast
      \end{itemize}
    } moreover {
      { assume CC: ss[]
        \begin{itemize}
          \item with SHFMT have \( \exists ss. ss[] \land sh=ss@r \) by blast
        \end{itemize}
      }
with CASE SPLIT2 SHFMT CC have ((r,ch),w,(r,c'))∈trcl (trss fg) by (auto intro: trss-rel)

moreover from SPLIT2 SHFMT EDGE have ((s,c),[e],[[],ch])∈trcl (trss fg) e#w=[e]@w

by simp

ultimately have ?case by blast

} ultimately have ?case ∨ ( ∃ ss. ss≠[] ∧ sh=ss@r ) by blast

moreover {

assume e≠LRet

with SPLIT1 IHP(3) have ( ∃ ss. ss≠[] ∧ sh=ss@r ) by (force elim!: trss.cases simp add: append-eq-Cons-conv)

moreover {

assume ( ∃ ss. ss≠[] ∧ sh=ss@r )

then obtain ss where CASE: ss≠[] sh=ss@r by blast

with SPLIT2 have ((ss@r, ch), w, r, c') ∈ trcl (trss fg) by simp

from IHP(1) OF this CASE(1) ] obtain wa wb ch' where IHAPP: w=wa@wb ((ss,ch),wa,([],ch'))∈trcl (trss fg) ((r,ch'),wb,(r,c'))∈trcl (trss fg) by blast

moreover from CASE SPLIT1 have ((s @ r, c), e, ss@r, ch) ∈ trss fg by simp

moreover from trss-stack-decomp-s(OF this IHP(3)) have ((s, c), e, ss, ch) ∈ trss fg by auto

with IHAPP have ((s, c), e#wa, ([]),ch') ∈ trcl (trss fg) by (rule-tac

trcl.cons)

moreover from IHAPP have e#wa=(e#wa)@wb by auto

ultimately have ?case by blast

} ultimately show ?case by blast

qed

lemma trss-return-cases[cases set]: !u r c. [((u≠r,c),w,(r',c')∈trcl (trss fg));

!! s' u. [ r'=s@u#r' ; (u),w,(s@u,c')∈trcl (trss fg) ] ⇒ P;

!! wa wb ch. [ w=wa@wb; ((u),w,([]),ch)∈trcl (trss fg) ; ((r),wb,(r',c')∈trcl (trss fg) ] ⇒ P]

⇒ P

proof (induct w rule: length-compl-induct)

case Nil thus ?case by auto

next

case (Cons e w) note IHP=this

then obtain sh ch where SPLIT1: ((u≠r,c),e,(sh,ch))∈trss fg and SPLIT2:

((sh,ch),w,(r',c')∈trcl (trss fg) by (fast dest: trcl-ancons)

{ fix ba q

assume CASE: e=LBase ba ∨ e=LSpawn q

with SPLIT1 obtain v where E: sh=v#r (((u),c),e,(v),ch))∈trss fg by (auto elim!: trss.cases intro: trss.intros)

})
with SPLIT2 have \((v\#r, ch), w, (r', c'))\) \(\in\) trcl (trss f g) by simp

hence \(?case proof\) (cases rule: IHP(1)[of w, simplified, cases set])

\(\text{case (1 s' w')}\) note CC=\(\text{this}\)

with \(E(2)\) have \(((\din u, c), e\#w, (s'\@([u'], c'))\) \(\in\) trcl (trss f g) by simp

from IHP(3)\[OF CC(1)\] this show \(?thesis\).

next

\(\text{case (2 wa wb ct)}\) note CC=\(\text{this}\)

with \(E(2)\) have \(((\din u, c), e\#wa, ([], ct))\) \(\in\) trcl (trss f g) \(e\#w = (e\#wa)\@wb\) by simp

all

from IHP(4)\[OF this (2, 1) CC(3)\] show \(?thesis\).

qed

\)

moreover

\(\text{assume CASE: e=LRet}\)

with SPLIT1 have \(\text{sh=}\(r\) \(((\din u, c), ([], ch))\) \(\in\) trcl (trss f g)\) by (auto elim!: trss.cases intro: trss.intros)

with IHP(4)\[OF - this (2)\] SPLIT2 have \(?case by auto\)

\)

moreover

\(\text{fix q}\)

\(\text{assume CASE: e=LCall q}\)

with SPLIT1 obtain \(u'\) where SHFMT: \(\text{sh=entry fg q} \# u' \# r\) \(((\din u, c), e, (entry f g q \# u', ch))\) \(\in\) trcl (trss f g) by (auto elim!: trss.cases intro: trss.intros)

with SPLIT2 have \(((\entry f g q \# u' \# r, ch), (r', c')) \in\) trcl (trss f g) by simp

hence \(?case proof\) (cases rule: IHP(1)[of w, simplified, cases set])

\(\text{case (1 st ut) note CC=}\(\text{this}\)

from trss-stack-comp[OF CC(2), where \(r=[u']\)] have \(((\entry f g q \# [u'], ch), w, (st \@ ([ut]) \@ [u'], c')) \in\) trcl (trss f g) by auto

with SHFMT(2) have \(((\din u, c), e\#w, (st \@ [ut]) \@ [u'], c') \in\) trcl (trss f g) by auto

from IHP(3)\[OF - this]\ CC(1) show \(?thesis by simp\)

next

\(\text{case (2 wa wb ct)}\) note CC=\(\text{this}\)

from trss-stack-comp[OF CC(2), where \(r=[u']\)] have \(((\entry f g q \# [u'], ch), wa, [u'], ct) \in\) trcl (trss f g) by simp

with SHFMT have PREPATH: \(((\din u, c), e\#wa, [u'], ct) \in\) trcl (trss f g) by simp

from CC have \(L\): length \(wb\leq\) length \(w\) by simp

from CC(3) show \(?case proof\) (cases rule: IHP(1)[of L, cases set])

\(\text{case (1 s'' u'')}\) note CCC=\(\text{this from trcl-concat[OF PREPATH CCC(2)]}\)

CC(1) have \(((\din u, c), e\#w, (s''\@([u''], c')) \in\) trcl (trss f g) by (simp)

from IHP(3)[OF CCC(1) \(\text{this}\)] show \(?thesis\).

next

\(\text{case (2 \(\text{wba wbb c'}\)) note CCC=}\(\text{this from trcl-concat[OF PREPATH CCC(2)]}\) CC(1) CCC(1)

\(\text{have e\#w = (e\#wba@wbb)@wbb ((\din u, c), e \# wba, [u'], c') \in\) trcl (trss f g) by auto}

from IHP(4)[OF this CCC(3)] show \(?thesis\).

qed

\)

ultimately show \(?case by (cases e, auto)\)

qed
lemma (in flowgraph) trss-find-call:
\[ !v r' c'. \left( \left( (\{sp\}, c), w, (v \# r', c') \right) \in \text{trcl} (\text{trss} fg) \; r' \neq \emptyset \right) \]
\[ \Rightarrow \exists rh ch p wa wb.
\]
\[ w = \text{wa} @ (\text{LCall} p) \# wb \wedge
\]
\[ \text{proc-of} fg v = p \wedge
\]
\[ ((\{sp\}, c), \text{wa}, (\text{rh}, \text{ch})) \in \text{trcl} (\text{trss} fg) \wedge
\]
\[ ((\text{rh}, \text{ch}), \text{LCall} p, ((\text{entry} fg p) \# r', \text{ch})) \in \text{trss} fg \wedge
\]
\[ ((\text{entry} fg p), \text{ch}), \text{wb}, (\{v\}, c')) \in \text{trcl} (\text{trss} fg)
\]

proof (induct w rule: length-compl-rev-induct)
case Nil thus \( \forall \text{case by (auto)} \)
next
case (snoc w e) note IHP=this
then obtain rh ch where SPLIT1: \( ((\{sp\}, c), w, (\text{rh}, \text{ch})) \in \text{trcl} (\text{trss} fg) \) and SPLIT2: \( ((\text{rh}, \text{ch}), e, (v \# r', c')) \in \text{trss} fg \) by (fast dest: trcl-rev-cons)

\[
\{
\]
\[ \begin{align*}
\text{assume } \exists u. \text{rh}=u \# r' \\
\text{then obtain u where RHFM}[\text{simp}]: \text{rh}=u \# r' \text{ by blast}
\end{align*}
\]
with SPLIT2 have \( \text{proc-of} fg u = \text{proc-of} fg v \) by (auto elim: trss.cases intro: edges-part)

moreover from IHP(1)[of w u r' ch, \( \text{OF - SPLIT1[simplified] IHP(3)} \)] obtain
rt ct p wa wb where

IHAPP: \( w = \text{wa} @ \text{LCall} p \# wb \text{ proc-of} fg u = p \) \( ((\{sp\}, c), \text{wa}, (\text{rt}, ct)) \in \text{trcl} (\text{trss} fg) \) \( ((\text{rt}, ct), \text{LCall} p, \text{entry} fg p \# r', ct) \in \text{trss} fg \)

\[ ((\text{entry} fg p), ct), \text{wb}, (\{u\}, ct)) \in \text{trcl} (\text{trss} fg) \text{ by (blast)}
\]
moreover
have \( ((\text{entry} fg p), ct), \text{wb}@[e], (\{v\}, c')) \in \text{trcl} (\text{trss} fg) \)
proof —

note IHAPP(3)
also from SPLIT2 have \( ((u), c, (\{v\}, c')) \in \text{trss} fg \) by (auto elim!: trss.cases intro!: trss.intros)

finally show \( \exists \text{thesis} . \)
qed
moreover from IHAPP have \( \text{wa}@[e] = \text{wa} @ \text{LCall} p \# (\text{wb}@[e]) \) by auto
ultimately have \( \exists \text{case by auto} \)
\}
moreover have \( \exists u. \text{rh}=u \# r' \lor \exists \text{case} \)
proof (rule trss.cases[OF SPLIT2, simp-all]) — Cases for base- and spawn edge are discharged automatically
— Case: call-edge

case (goal1 ca p r u vv) with SPLIT1 SPLIT2 show \( \exists \text{case by fastforce} \)
next
— Case: return edge

case (goal2 q r ca) note CC=this
hence [simp]: \( \text{rh}=(\text{return} fg q) \# v \# r' \) by simp
with IHP(1)[of w (return fg q) \( v \# r' \) ch, \( \text{OF - SPLIT1[simplified]} \)] obtain
rt ct wa wb where

IHAPP: \( w = \text{wa} @ \text{LCall} q \# wb \) \( ((\{sp\}, c), \text{wa}, (\text{rt}, ct)) \in \text{trcl} (\text{trss} fg) \) \( ((\text{rt}, ct), \text{LCall} q, \text{entry} fg q \# v \# r', ct) \in \text{trss} fg \)
Lemma (in flowgraph) trss-find-call:
assumes A: 
and EX: 
shows P
proof –
from trss-find-call[of A] obtain rh ch wa wb where FC:

moreover from FC(3) obtain uh where ADD: rh=[uh] (uh, Call p, u') \in edges
lemma (in flowgraph) trss-bot-proc-const:

\[ !s' a' c'. ((s@u], e), w, (s'@u', c')) \in \text{trcl} \Rightarrow \text{proc-of}\ fg\ u = \text{proc-of}\ fg\ u' \]

proof (induct \( w \) rule: rev-induct)

  case Nil thus \( ?\) case by auto

  next
  case (snoc \( e\) \( w\)) note IHP=this then obtain \( sh\ ch\) where SPLIT1: \( ((s@u], e), w, (sh, ch)) \in \text{trcl} \Rightarrow \text{proc-of}\ fg\ u = \text{proc-of}\ fg\ uh \)

  from SPLIT2 have \( sh \neq [] \) by (auto elim!: trss.cases)

  then obtain \( ssh\ uh\) where SHFMT: \( sh = ssh@c]uh\) by (fast dest: list-rev-decomp)

  with IHP(1)[of \( ssh\ uh\) \( ch\)] SPLIT1 have proc-of \( fg\ u = \text{proc-of}\ fg\ uh\) by auto

  also from SPLIT2 SHFMT have proc-of \( fg\ uh = \text{proc-of}\ fg\ u'\) by (cases rule: trss.cases) \( ch\) simp add: edges-part)+

  finally show \( ?\) case .

qed

— Specialized version of \text{flowgraph},trss-bot-proc-const that comes in handy for precision proofs of constraint systems

lemma (in flowgraph) trss-end-path-proc-const:

\[ \exists w' p. w = w'c]LRet\ (trcl\ (trss\ fg)) \Rightarrow \text{proc-of}\ fg\ p = q \]

using trss-bot-proc-const[of \( []\) entry \( fg\ p\) \( c\)\,\w,\( ]\return\ fg\ q\) \( c'\)\,\w]\in\text{trcl}\ (trss\ fg)\ \Rightarrow\ p = q,\ simplified] .

lemma \text{trss-2empty-to-2return}:

\[ \exists w' p. w = w'c]LRet\ (trcl\ (trss\ fg)) \Rightarrow \text{proc-of}\ fg\ p = q \]

proof –

  assume \( A\): \( ((s, c), w, ([]], c')) \in \text{trcl} \Rightarrow \text{proc-of}\ fg\ s \neq [] \)

  hence \( w \neq []\) by auto

  then obtain \( w' e\) where WD: \( w = w'e[]\) by (fast dest: list-rev-decomp)

  with \( A\) obtain \( sh\ ch\) where SPLIT: \( ((s, c), w', (sh, ch)) \in \text{trcl} \Rightarrow (sh, ch, e, ([]], c')) \in \text{trss}\ fg\ by\ (fast\ dest: \text{trcl-rev-uncons})\)

  from SPLIT(1) obtain \( p\) where \( e = LRet\ sh = ]\return\ fg\ p\) \( ch = c'\) by (cases rule: trss.cases, auto)

  with SPLIT(1) WD show \( ?\)thesis by blast

qed

lemma \text{trss-2return-to-2empty}:

\[ ((s, c), w, ([return\ fg\ p], c')) \in \text{trcl} \Rightarrow \text{proc-of}\ fg\ s \neq [] \]

apply (subgoal-tac \( ([\text{return}\ fg\ p], c'), LRet, ([]], c'))\in\text{trss}\ fg\ by\ (auto\ dest: \text{trcl-rev-cons} intro: \text{trss.intros})

7.7.2 Adding threads

lemma \text{trss-env-increasing-s}:

\[ ((s, c), e, (s', c')) \in \text{trss}\ fg \Rightarrow c \leq s' \]

by (auto elim!: trss.cases)

lemma \text{trss-env-increasing}:

\[ ((s, c), w, (s', c')) \in \text{trcl} \Rightarrow c \leq s' \]

47
by (induct rule: trcl-pair-induct) (auto dest: trss-env-increasing-s order-trans)

7.7.3 Conversion between environment and monitor restrictions

**Lemma** trss-mon-c-no-ctx:

\[\{(s, e, (s', c')) \in \text{trss } fg \mid mon-c \ fg \ e \land mon-c \ fg \ c = \{\}\}\]

by (erule trss.cases) auto

**Lemma** (in flowgraph) trss-mon-w-no-ctx:

\[\{(s, w, (s', c')) \in \text{trcl } (\text{trss } fg) \mid mon-w \ fg \ w \land mon-c \ fg \ c = \{\}\}\]

by (induct rule: trcl-pair-induct) (auto dest: trss-mon-e-no-ctx simp add: trss-c-no-mon-s)

**Lemma** (in flowgraph) trss-modify-context-s:

\[\exists cn. \{(s, e, (s', c')) \in \text{trss } fg \mid mon-e \ fg \ e \land mon-c \ fg \ cn = \{\}\}\]

\[\Rightarrow \exists csp. \ c' = csp + c \land mon-c \ fg \ csp = \{\}\ \land\ \{(s, cn), w, (s', csp + cn)\} \in \text{trss } fg\]

by (erule trss.cases) (auto intro!: trssintros)

**Lemma** (in flowgraph) trss-modify-context-s [rule-format]:

\[\{(s, w, (s', c')) \in \text{trcl } (\text{trss } fg)\}\]

\[\Rightarrow \forall cn. \ mon-w \ fg \ w \land mon-c \ fg \ cn = \{\}\]

\[\Rightarrow \exists csp. \ c' = csp + c \land mon-c \ fg \ csp = \{\}\ \land\ \{(s, cn), w, (s', csp + cn)\} \in \text{trcl } (\text{trss } fg)\]

**Proof** (induct rule: trcl-pair-induct)

**Case** empty thus \(\text{?case by simp}\)

**Next**

**Case** (cons \(s \ c \ e \ sh \ w \ s' \ c'\)) note IHP=this show \(\text{?case}\)

**Proof** (intro allI impI)

fix \(cn\)

assume MON: mon-w \(fg \ (e \neq w) \land mon-c \ fg \ cn = \{\}\)

from trss-modify-context-s [OF IHP(1)] MON obtain csph where S1: \(ch = csph + c \land mon-c \ fg \ csph = \{\}\ \{(s, cn), e, sh, csph + cn\} \in \text{trss } fg\) by auto

with MON have mon-w \(fg \ w \land mon-c \ fg \ (csph + cn) = \{\}\) by (auto simp add: mon-c-unconc)

with IHP(3) [rule-format] obtain csp where S2: \(c' = csp + ch \land mon-c \ fg \ csp = \{\}\ \{(s, sh, csph + cn)\} \in \text{trcl } (\text{trss } fg)\) by blast

from S1 S2 have \(c' = (\text{csph + ch}) + c \land mon-c \ fg \ (\text{csph + ch}) = \{\}\ \{(s, cn), e \neq w, s', csph + cn\} \in \text{trcl } (\text{trss } fg)\) by (auto simp add: union-assoc mon-c-unconc)

thus \(\exists csp. \ c' = csp + c \land mon-c \ fg \ csp = \{\}\ \land\ \{(s, cn), e \neq w, s', csph + cn\}\)

\(\in \text{trcl } (\text{trss } fg)\) by blast

qed

**Lemma** trss-add-context-s:

\[\{(s, e, (s', c')) \in \text{trss } fg \mid mon-e \ fg \ e \land mon-c \ fg \ ce = \{\}\}\]

\[\Rightarrow \{(s, c + ce), (s', c' + ce)\} \in \text{trss } fg\]

by (auto elim!: trss.cases intro!: trssintros simp add: union-assoc mon-c-unconc)

**Lemma** trss-add-context:

\[\{(s, w, (s', c')) \in \text{trcl } (\text{trss } fg) \mid mon-w \ fg \ w \land mon-c \ fg \ ce = \{\}\}\]

\[\Rightarrow \{(s, c + ce), w, (s', c' + ce)\} \in \text{trcl } (\text{trss } fg)\]

48
proof (induct rule: trcl-pair-induct)
  case empty thus \?case by simp
next
  case (cons s c e sh ch w s' c') note IHP=this
  from IHP(4) have MM: mon-e fg e ∩ mon-c fg ce = {} mon-w fg w ∩ mon-c fg ce = {} by auto
  from trcl.cons[OF trss-add-context-s[OF IHP(1) MM(1)] IHP(3)[OF MM(2)]]
  show \?case .
qed

lemma trss-drop-context-s: \[((s,c+ce),e,(s',c'+ce))\in trss fg\] \[\implies ((s,c),e,(s',c'))\in trss fg \land mon-e fg e \cap mon-c fg ce = {}\]
by (erule trss, cases) (auto intro: trss.intros simp add: mon-c-unconc union-assoc[of - c ce, symmetric])

lemma trss-drop-context: \!\!s c. \[((s,c+ce),w,(s',c'+ce))\in trcl (trss fg)\] \[\implies ((s,c),w,(s',c'))\in trcl (trss fg) \land mon-w fg w \cap mon-c fg ce = {}\]
proof (induct w)
  case Nil thus \?case by auto
next
  case (Cons e w) note IHP=this
  then obtain sh ch where SPLIT: \[((s,c+ce),e,(sh,ch))\in trss fg ((sh,ch),w,(s',c'+ce))\in trcl (trss fg)\] by (fast dest: trcl-unscons)
  from trss-c-fmt-s[OF SPLIT(1)] obtain csp where CHFMT: ch = (csp + c) + ce by (auto simp add: union-assoc)
  from CHFMT trss-drop-context-s SPLIT(1) have \[((s,c),e,(sh,csp+c))\in trss fg mon-e fg e \cap mon-c fg ce = {}\] by blast+
  moreover from CHFMT IHP(1) SPLIT(2) have \[((sh,csp+c),w,(s',c'))\in trcl (trss fg) mon-w fg w \cap mon-c fg ce = {}\] by blast+
  ultimately show \?case by auto
qed

lemma trss-xchange-context-s:
  assumes A: \[((s,c),e,(s',csp+c))\in trss fg\] and M: mon-c fg cn \subseteq mon-c fg e
  shows \[((s,cn),e,(s',csp+cn))\in trss fg\]
proof
  from trss-drop-context-s[of - \{\#\}, simplified, OF A] have DC: \[((s, \{\#\}), e, s', csp)\in trss fg mon-c fg e \cap mon-c fg c = {}\] by simp-all
  with M have mon-e fg e \cap mon-c fg cn = {} by auto
  from trss-add-context-s[OF DC(1) this] show \?thesis by auto
qed

lemma trss-xchange-context:
  assumes A: \[((s,c),w,(s',csp+c))\in trcl (trss fg)\] and M: mon-c fg cn \subseteq mon-c fg e
  shows \[((s,cn),w,(s',csp+cn))\in trcl (trss fg)\]
proof
  from trss-drop-context[of - \{\#\}, simplified, OF A] have DC: \[((s, \{\#\}), w, s',

49
\[ \text{csp} \in \text{trclneapolis} \text{fg} \text{mon-w fg} \text{w} \cap \text{mon-c fg} \text{c} = \{\} \text{ by simp-all} \]

with \( M \) have \( \text{mon-w fg} \text{w} \cap \text{mon-c fg} \text{c} = \{\} \text{ by auto} \)

from \( \text{trss-add-context}[\text{OF DC(1)}] \) show ?thesis by auto

qed

lemma \( \text{trss-drop-all-context-s}[\text{cases set, case-names dropped}] \):

assumes \( A: ((s,c),e,(s',c'))\in\text{trss fg} \)

and \( C: \forall \text{csp}. \; [c'=?\text{csp}+c; ((s,\{\#\}),e,(s',\text{csp}))\in\text{trss fg}] \implies P \)

shows \( P \)

using \( A \) proof (cases rule: \( \text{trss-cases-s} \))

\begin{itemize}
  \item case \( \text{no-spawn} \) with \( \text{trss-xchange-context-s}[\text{of s c e s'}\{\#\} fg \{\#\}] \) \( A \) \( C \) show \( P \) by auto
  \item case \( (\text{spawn p u v}) \) with \( \text{trss-xchange-context-s}[\text{of s c e s'}\{\#\} entry fg p\{\#\} fg \{\#\}] \) \( A \) \( C \) show \( P \) by auto
\end{itemize}

qed

lemma \( \text{trss-drop-all-context}[\text{cases set, case-names dropped}] \):

assumes \( A: ((s,c),w,(s',c'))\in\text{trcl (trss fg)} \)

and \( C: \forall \text{csp}. \; [c'=?\text{csp}+c; ((s,\{\#\}),w,(s',\text{csp}))\in\text{trcl (trss fg})] \implies P \)

shows \( P \)

using \( A \) proof (cases rule: \( \text{trss-cases} \))

\begin{itemize}
  \item case \( (c-case csp) \) with \( \text{trss-xchange-context-s}[\text{of s c w s'}\{\#\} csfg \{\#\}] \) \( A \) \( C \) show \( P \) by auto
\end{itemize}

qed

lemma \( \text{tr-add-context-s} \):

\[ [ (c,e,c')\in tr \; fg; \; \text{mon-e fg} \; e \cap \text{mon-c fg} \; ce = \{\} ] \implies (c+ce,e,c'+ce)\in tr \; fg \]

by (erule \( \text{gtrE} \)) (auto simp add: \( \text{mon-c-unconc union-assoc intro: gtrI-s dest: trss-add-context-s} \))

lemma \( \text{tr-add-context} \):

\[ [ (c,w,c')\in trcl \; (tr \; fg); \; \text{mon-w fg} \; w \cap \text{mon-c fg} \; ce = \{\} ] \implies (c+ce,w,c'+ce)\in trcl \; (tr \; fg) \]

proof (induct rule: \( \text{trcl.induct} \))

case empty thus ?case by auto

next

\begin{itemize}
  \item case \( (\text{cons c e c'} \; w \; c'')\) note \( \text{IHP=this} \)
  \item from \( \text{tr-add-context-s}[\text{OF IHP(1), of ce}] \) \( \text{IHP(4)} \) have \( (c + ce, e, c' + ce) \in tr \; fg \) by auto
  \item also from \( \text{IHP(3,4)} \) have \( (c' + ce, w, c'' + ce) \in trcl \; (tr \; fg) \) by auto
\end{itemize}

finally show ?case .

qed

end

\section{Normalized Paths}

theory \( \text{Normalization} \)

50
The idea of normalized paths is to consider particular schedules only. While the original semantics allows a context switch to occur after every single step, we now define a semantics that allows context switches only before non-returning calls or after a thread has reached its final stack. We then show that this semantics is able to reach the same set of configurations as the original semantics.

8.1 Semantic properties of restricted flowgraphs

It makes the formalization smoother, if we assume that every thread’s execution begins with a non-returning call. For this purpose, we defined syntactic restrictions on flowgraphs already (cf. Section 6.3). We now show that these restrictions have the desired semantic effect.

— Procedures with isolated return nodes will never return

**Lemma (in eflowgraph) iso-ret-no-ret: !!u c. [**

\[
\begin{align*}
& \text{isolated-ret}\ fg\ p; \\
& \text{proc-of}\ fg\ u = p; \\
& u \neq \text{return}\ fg\ p; \\
& (([u],c),w,([\text{return}\ fg\ p',c'])) \in \text{trcl}\ (\text{trss}\ fg) \\
& \implies \text{False}
\end{align*}
\]

**Proof (induct w rule: length-compl-induct)**

**case** Nil **thus** ?case **by** auto

**next**

**case** (Cons e w) **note** IHP=this

then obtain sh ch where SPLIT1: (([u],c),e,(sh,ch)) \in \text{trss}\ fg\ and SPLIT2: ((sh,ch),w,([\text{return}\ fg\ p',c'])) \in \text{trcl}\ (\text{trss}\ fg) **by** (fast dest: trcl-uncons)

show ?case **proof** (cases e)

**case** LRet with SPLIT1 IHP(3,4) **show** False **by** (auto elim!: trss_cases)

**next**

**case** LBase with SPLIT1 IHP(2,3) **obtain** v where A: sh=[v] \proc-of\ fg\ v = p \not\eq\text{return}\ fg\ p\ by\ (force\ elim!:\ trss_cases simp add: edges-part isolated-ret-def)

with IHP SPLIT2 **show** False **by** auto

**next**

**case** (LSpawn q) with SPLIT1 IHP(2,3) **obtain** v where A: sh=[v] \proc-of\ fg\ v = p \not\eq\text{return}\ fg\ p\ by\ (force\ elim!:\ trss_cases simp add: edges-part isolated-ret-def)

with IHP SPLIT2 **show** False **by** auto

**next**

**case** (LCall q) with SPLIT1 IHP(2,3) **obtain** uh where A: sh=entry\ fg\ q#[uh] \proc-of\ fg\ uh = p \not\eq\text{return}\ fg\ p\ by\ (force\ elim!:\ trss_cases simp add: edges-part isolated-ret-def)

with SPLIT2 **have** B: ((entry\ fg\ q#[uh],ch),w,([\text{return}\ fg\ p',c'])) \in \text{trcl}\ (\text{trss}\ fg) **by** simp

from trss-return-cases[of B] **obtain** wI w2 ct where C: w=wI@w2 length w2 \leq\ length\ w\ ((entry\ fg\ q#,ch),wI,([],ct)) \in \text{trcl}\ (\text{trss}\ fg)\ (([uh],ct),w2,([\text{return}\ fg\ p',c'])) \in \text{trcl}\ (\text{trss}\ fg) **by** (auto)
from IHP(1) [OF C(2)] IHP(2) A(2,3) C(4)] show False .

qed

— The first step of an initial procedure is a call


— There are no same-level paths starting from the entry node of an initial procedure


— There are no same-level or returning paths starting from the entry node of an initial procedure
from no-sl-from-initial[OF this A(2) B(2)] show False .

qed

next

case (Cons u rr) with A(4) have r' = [u] by auto

with no-sl-from-initial[OF A(1,2)] A(3) show False by auto

qed

8.2 Definition of normalized paths

In order to describe the restricted schedules, we define an operational semantics that performs an atomically scheduled sequence of steps in one step, called a macrostep. Context switches may occur after macrosteps only. We call this the normalized semantics and a sequence of macrosteps a normalized path.

Since we ensured that every path starts with a non-returning call, we can define a macrostep as an initial call followed by a same-level path of the called procedure. This has the effect that context switches are either performed before a non-returning call (if the thread makes a further macrostep in the future) or after the thread has reached its final configuration.

As for the original semantics, we first define the normalized semantics on a single thread with a context and then use the theory developed in Section 5 to derive interleaving semantics on multisets and configurations with an explicit local thread (loc/env-semantics, cf. Section 5.4).

inductive-set

ntrs :: ('n', 'p', 'ba', 'm', 'more) flowgraph-rec-scheme ⇒

((('n list × 'n conf) × ('p', 'ba) label list × ('n list × 'n conf)) set

for fg

where

— A macrostep transforms one thread by first calling a procedure and then doing a same-level path

ntrs-step: ∃((u # r, ce), LCall p, (entry fg p # u' # r, ce))∈trss fg;
       ∃((entry fg p[1], ce), w, ([v[2], ce'])∈trcl (trss fg)) ⇒
       ∃((u # r, ce), LCall p # w, (v # u' # r, ce'))∈ntrs fg

abbreviation ntr where ntr fg == gtr (ntrs fg)

abbreviation ntrp where ntrp fg == gtrp (ntrs fg)

interpretation ntrs: env-no-step ntrs fg
apply (rule env-no-step.intro)
apply (erule ntrs.cases)
apply clarsimp
apply (erule trss-c-cases)

2Same-level paths are paths with balanced calls and returns. The stack-level at the beginning of their execution is the same as at the end, and during the execution, the stack never falls below the initial level.
apply auto
done

8.3 Representation property for reachable configurations

In this section, we show that a configuration is reachable if and only if it is reachable via a normalized path.

The first direction is to show that a normalized path is also a path. This follows from the definitions. Note that we first show that a single macrostep corresponds to a path and then generalize the result to sequences of macrosteps.

**Lemma** ntr-is-trss-s: \(((s,c),w,(s',c'))\) ∈ ntrs \(\Rightarrow\) \(((s,c),w,(s',c'))\) ∈ trcl \(\text{trss}\)

**Proof** (erule ntrs_cases, auto)

1. Assume \(A\): \(((u \neq r, c), LCall p, entry fg p \neq u' \neq r, c)\) ∈ trss \(\Rightarrow\) \(((entry fg p \neq u' \neq r, c), w, v \neq u' \neq r, c')\) ∈ trcl \(\text{trss}\) by simp

2. From trss-stack-comp[OF A(2), of u'#r] have \(((entry fg p \neq u' \neq r, c), w, v \neq u' \neq r, c')\) ∈ trcl \(\text{trss}\) by simp

3. With A(1) show \(((u \neq r, c), LCall p \neq w, v \neq u' \neq r, c')\) ∈ trcl \(\text{trss}\) by auto

**QED**

**Lemma** ntr-is-trss: \(((s,c),w,(s',c'))\) ∈ trcl \(\text{ntrs}\)

\(\Rightarrow\) \(((s,c),\text{foldl } (\text{op }@) \emptyset w, (s',c'))\) ∈ trcl \(\text{trss}\)

**Proof** (induct rule: trcl_pair_induct)

1. Case empty thus \(\text{auto}\) simp

2. Next

3. Case (cons c e c e c' c) note IHP=this

4. From trcl-concat[OF ntr-is-trss-s[OF IHP(1)] IHP(3)] foldl-conc-empty-eq[of e w] show \(\text{auto}\)

**QED**

**Lemma** ntr-is-tr: \((c,w,c')\) ∈ ntr \(\Rightarrow\) \((c,w,c')\) ∈ trcl \(\text{tr}\)

**Proof** (induct rule: trcl_induct)

1. Case empty thus \(\text{auto}\)

2. Next

3. Case (cons c e c' c w c') note IHP=this

4. From trcl-concat[OF ntr-is-tr-s[OF IHP(1)] IHP(3)] foldl-conc-empty-eq[of e w] show \(\text{auto}\)

**QED**

The other direction requires to prove that for each path reaching a configuration there is also a normalized path reaching the same configuration. We need an auxiliary lemma for this proof, that is a kind of append rule: *Given a normalized path reaching some configuration c, and a same level...*
or returning path from some stack in c, we can derive a normalized path to c modified according to the same-level path. We cannot simply append the same-level or returning path as a macrostep, because it does not start with a non-returning call. Instead, we will have to append it to some macrostep in the normalized path, i.e. move it „left” into the normalized path.

Intuitively, we can describe the concept of the proof as follows: Due to the restrictions we made on flowgraphs, a same-level or returning path cannot be the first steps on a thread. Hence there is a last macrostep that was executed on the thread. When this macrostep was executed, all threads held less monitors then they do at the end of the execution, because the set of monitors held by every single thread is increasing during the execution of a normalized path. Thus we can append the same-level or returning path to the last macrostep on that thread. As a same-level or returning path does not allocate any monitors, the following macrosteps remain executable. If we have a same-level path, appending it to a macrostep yields a valid macrostep again and we are done. Appending a returning path to a macrostep yields a same-level path. In this case we inductively repeat our argument.

The actual proof is strictly inductive; it either appends the same-level path to the last macrostep or inductively repeats the argument.

**Lemma (in flowgraph)** ntr-sl-move-left: \( !ce u r w r' ce' \).

\[ ((\#[entry fg p]\#),wu,\{\# u#r \#\}+ce) \in trcl (ntr fg); \]

\[ ((\[u],ce),w,(r',ce')) \in trcl (trss fg); \]

\[ \text{initialproc } fg p; \]

\[ \text{length } r' \leq 1; \text{ w} \neq [] \]

\[ \implies \exists wu'. ((\#[entry fg p]\#), wu',\{\# r''@r \#\}+ce') \in trcl (ntr fg) \]

**Proof** (induct \( uu \) rule: rev-induct)

**Case Nil** note **CC=This**

**Case u=entry fg p** by auto

— If the normalized path is empty, we get a contradiction, because there is no same-level path from the initial configuration of a thread

**With** **CC(2) no-retsl-from-initial**([OF **CC(5,3) - CC(4)**]) have **False** by blast

thus ?case ..

**Next**

**Case (snoc ee uu)** note **IHP=This**

— In the induction step, we extract the last macrostep

**Then obtain** \( ch \) where **SPLIT**: \( ((\#[entry fg p]\#),ww,\text{ch}) \in trcl (ntr fg); (ch,ce,\{\# u#r \#\}+ce) \in ntr fg \) by (fast dest: trcl-rev-uncons)

— The last macrostep first executes a call and then a same-level path

**From** **SPLIT(2)** obtain \( q wus uh rh ceh uh' vt cet \) where

**STEPFMT**: \( ee = \text{LCall } q\#wus \text{ ch} = (\# \text{ uh#rh \#}) + ceh \) \( \{\# \text{ vt#uh'\#rh \#}\} + \text{ce} = \{\# \text{ vt#uh'\#rh \#}\} + \text{ce} \) = \( (\text{uh#rh,ceh}) \), \( \text{LCall } q, (\text{entry fg } q\#\text{uh'\#rh,ceh}) \in trss fg \)

\( (((\text{entry fg } q),\text{ceh}),\text{wus},(\text{vt},\text{cet})) \in trcl (trss fg) \)

by (blast elim!: gtrE ntr.scs[ simplified ])

— Make a case distinction whether the last step was executed on the same thread as the sl/ret-path or not

**From** **STEPFMT(3)** show ?case **proof** (cases rule: mset-single-cases)

— If the sl/ret path was executed on the same thread as the last macrostep
case loc note CASE=this hence C": u=vt r=uh’#rh ce=ctet by auto
— we append it to the last macrostep.

with STEPFMT(5) IHP(3) have NEWPATH: ([(entry fg q), ceh], wws@w, (r’, ce’)) ∈ trcl
(trss fg) by (simp add: trcl-concat)
— We then distinguish whether we appended a same-level or a returning path

show ?thesis proof (cases r’)
— If we appended a same-level path

case (Cons v’) — Same-level path with IHP(5) have CC: r’=[v’] by auto
— The macrostep still ends with a same-level path

with NEWPATH have (((entry fg q), ceh), wws@w, (v’, ce’)) ∈ trcl (trss fg) by simp
— and thus remains a valid macrostep

from gtrI-s[OF ntrs-step[OF STEPFMT(4), simplified, OF this]] have (\{ #u # rh # \} + ceh, LCall q # wws@w, \{ #v’ # uh’ # rh # \} + ce’) ∈ ntr fg .
— that we can append to the prefix of the normalized path to get our proposition

with STEPFMT(2) SPLIT(1) CC C’(2) have (\{ #entry fg p \} #, wws@[LCall q#wws@w, # r’@r #]) + ce’ ∈ trcl (ntr fg) by (auto simp add: trcl-rev-cons)
thus ?thesis by blast
next
— If we appended a returning path

case Nil note CC=this
— The macrostep now ends with a returning path, and thus gets a same-level path

have NEWSL: (\{ uh, ceh \}, LCall q # wws @ w, \{ uh’, ce’ \}) ∈ trcl (trss fg)
proof —

from STEPFMT(4) have (\{ uh, ceh \}, LCall q,(entry fg q#[uh’, ceh])∈trss fg by (auto elim!: trss.cases intro: trss.intros)
also from trss-stack-comp[OF NEWPATH] CC have ((entry fg q#[uh’, ceh], wws@w,(\{ uh’, ce’ \}))∈trcl (trss fg) by auto

finally show ?thesis .

qed
— Hence we can apply the induction hypothesis and get the proposition

from IHP(1)(OF - NEWSL] SPLIT STEPFMT(2) IHP(4) CC C’(2) show ?thesis by auto

qed
next
— If the sl/ret path was executed on a different thread than the last macrostep

case (env cc) note CASE=this
— we first look at the context after the last macrostep. It consists of the threads
that already have been there and the threads that have been spawned by the last macrostep

from STEPFMT(5) obtain cspt where CETFMT: cet=cspt+ceh !s. s:#cspt
⇒ ∃ p. s=[entry fg p] ∧ initialproc fg p
by (unfold initialproc-def) (erule trss-cases, blast)
— The spawned threads do not hold any monitors yet

hence CSPT-NO-MON: mon-c fg cspt = {} by (simp add: c-of-initial-no-mon)
— We now distinguish whether the sl/ret path is executed on a thread that was
just spawned or on a thread that was already there

56
from CASE(1) CETFMT(1) have u#r : cspt+ceh by auto
thus ?thesis proof (cases rule: mset-un-cases[cases set])
— The sl/ret path cannot have been executed on a freshly spawned thread
due to the restrictions we made on the flowgraph
case left — Thread was spawned with CETFMT obtain q where u=entry
fg q r = [] initialproc fg q by auto
with IHP(3,5,6) no-retsl-from-initial have False by blast
thus ?thesis ..
next
— Hence let’s assume the sl/ret path is executed on a thread that was already
there before the last macrostep
case right note CC=this
— We can write the configuration before the last macrostep in a way that one
sees the thread that executed the sl/ret path
hence CEHFMT: ceh=\{ # u#r \#\}+(ceh-\{ # u#r \#\}) by auto
have CHFMT: ch = \{ # u#r \#\} + (\{ # uh#rh \#\}+(ceh-\{ # u#r \#\}))
proof —
from CEHFMT STEPFMT(2) have ch = \{ # uh#rh \#\} + (\{ # u#r
\#\}+(ceh-\{ # u#r \#\})) by simp
thus ?thesis by (auto simp add: union-ac)
qed
— There are not more monitors than after the last macrostep
have MON-CE: mon-c fg (\{ # uh#rh \#\}+(ceh-\{ # u#r \#\})) \subseteq mon-c fg
ce proof —
have mon-n fg uh \subseteq mon-n fg uh’ using STEPFMT(4) by (auto elim!: trss.cases dest: mon-n-same-proc edges-part)
moreover have mon-c fg (ceh-\{ # u#r \#\}) \subseteq mon-c fg cc proof —
from CASE(3) CETFMT have cc=(cspt+ceh)-\{ # u#r \#\} by simp
with CC have cc = cspt+(ceh-\{ # u#r \#\}) by (auto simp add: diff-union-single-conv)
with CSPT-NO-MON show ?thesis by (auto simp add: mon-c-unconc)
qed
ultimately show ?thesis using CASE(2) by (auto simp add: mon-c-unconc)
qed
— The same-level path preserves the threads in its environment and the threads
that it creates hold no monitors
from IHP(3) obtain cspt’ where CE’FMT: ce’=cspt’+cc mon-c fg cspt’ = {} by (—) (crule trss-cases, blast intro: c-of-initial-no-mon)
— We can execute the sl/ret-path also from the configuration before the last step
from trss-xchange-context[OF - MON-CE] IHP(3) CE’MT have NSL:
(((\{u\}, \{ #uh # rh\#\} + (ceh - \{ #u # r\#\})), w, r’, cspt’ + (\{ #uh # rh\#\} + (ceh
- \{ #u # r\#\}))) \in trcl (trss fg) by auto
— And with the induction hypothesis we get a normalized path
from IHP(1)[OF - NSL IHP(4,5,6)] SPLIT(1) CHFMT obtain ww’ where
NNPATH: (\{ #entry fg p\#\}, ww’, \{ #r’ @ r\#\} + (cspt’ + ((#uh # rh\#\} + (ceh
- \{ #u # r\#\}))) \in trcl (ntr fg) by blast
— We now show that the last macrostep can also be executed from the new
configuration, after the sl/ret path has been executed (on another thread)
have ((#r’ @ r\#\} + (cspt’ + ((#uh # rh\#\} + (ceh - \{ #u # r\#\})), ee,
\{\#vt \neq \#uh' \neq \#rh\}\} + (csp' + (\{\#r' @ \#r\} + (ceh - \{\#u \neq \#r\})))\}
\in ntr fg

\textbf{proof} –
— This is because the sl/ret path has not allocated any monitors
have \textbf{MON-CEH}: mon-c fg (\{\#r' @ \#r\} + (csp' + (ceh - \{\#u \neq \#r\})))
\subseteq mon-c fg ceh \textbf{proof} –
from IHP(3,5) trss-bot-proc-const[of [] u ce w [] - ce'] mon-n-same-proc
have mon-s fg r' \subseteq mon-n fg u by (cases r') (simp, force)
moreover from CEHFMT have mon-c fg ceh = mon-c fg (\{\#u \neq \#r\} + (ceh - \{\#u \neq \#r\}))

\textbf{ultimately show} \textit{thesis} using CE'FMT(2) by (auto simp add: mon-c-unconc mon-s-unconc)
\textbf{qed}

— And we can reassemble the macrostep within the new context
\textbf{note} trss-xchange-context-s[of - MON-CEH, where csp=\{\#\}, simplified, OF STEPFMT(4)]
moreover from trss-xchange-context[of - MON-CEH, of \{entry fg q\} wusu
\{vt\} csp\} STEPFMT(5) CETFMT(1) have
((\{entry fg q\}, \{\#r' @ \#r\} + (csp' + (ceh - \{\#u \neq \#r\}))), wusu, \{vt\},
csp' + (\{\#r' @ \#r\} + (csp' + (ceh - \{\#u \neq \#r\})))) \in trcl (trss fg) by blast
moreover \textbf{note} STEPFMT(1)
ultimately have ((uh#rh, (\{\#r' @ \#r\} + (csp' + (ceh - \{\#u \neq \#r\}))))\}, \{vt\#/uh'#/rh, csp'+ (\{\#r' @ \#r\} + (csp' + (ceh - \{\#u \neq \#r\}))))\}\in ntrsl
fg by (blast intro: ntrsl.intros|simplified)
from gtrl-s[of this] show \textit{thesis} by (simp add: union-ac)
\textbf{qed}

— Finally we append the last macrostep to the normalized paths we obtained
by the induction hypothesis
from trcl-rev-cons[of NNPATH this] have (\{\#\entry fg p\#\}, wusu @ \{ce\},
\{\#vt \neq \#uh' \neq \#rh\}\} + (csp' + (\{\#r' @ \#r\} + (csp' + (ceh - \{\#u \neq \#r\}))))
\subseteq trcl (ntr fg) .
— And show that we got the right configuration
moreover from CC CETFMT CASE(3)[symmetric] CASE(2) CE'FMT(1)
have (\#vt \neq \#uh' \neq \#rh\} + (csp' + (\{\#r' @ \#r\} + (csp' + (ceh - \{\#u \neq \#r\})))) = (\# r'@r') + ce' by (simp add: union-ac diff-union-single-convs)

\textbf{ultimately show} \textit{thesis} by \textbf{auto}
\textbf{qed}
\textbf{qed}
\textbf{qed}

Finally we can prove: \textit{Any reachable configuration can also be reached by a normalized path.} With eflowgraph.ntr-sl-move-left we can easily show this lemma With eflowgraph.ntr-sl-move-left we can easily show this by induction on the reaching path. For the empty path, the proposition follows trivially. Else we consider the last step. If it is a call, we can execute it as a macrostep and get the proposition. Otherwise the last step is a same-level (Base, Spawn) or returning (Ret) path of length 1, and we can append it to the normalized path using eflowgraph.ntr-sl-move-left.
lemma (in eflograph) normalize: []
(cstart, w.c') ∈ trcl (tr fg);
cstart={# [entry fg p] #};
initialproc fg p []
⇒ ∃ w'. ((# [entry fg p] #), w', c') ∈ trcl (ntr fg)
— The lemma is shown by induction on the reaching path
proof (induct rule: trcl-rev-induct)
— The empty case is trivial, as the empty path is also a valid normalized path

next

case (snoc cstart w c e c') note IHP=this
— In the inductive case, we can assume that we have an already normalized path
and need to append a last step

then obtain w' where IHP': ((# [entry fg p] #), w', c') ∈ trcl (ntr fg) (c, e, c') ∈ tr
fg by blast
— We make explicit the thread on that this last step was executed
from gtr-find-thread[OF IHP'(2)] obtain s ce s' ce' where TSTEP: c = {# s#} + ce c' = {# s' #} + ce' ((s, ce), e, (s', ce')) ∈ trss fg by blast
— The proof is done by a case distinction whether the last step was a call or not

{ — Last step was a procedure call
  fix q
  assume CASE: e=LCall q
  — As it is the last step, the procedure call will not return and thus is a valid
macrostep
  have (c, LCall q # [], e') ∈ ntr fg using TSTEP CASE by (auto elim!: trss.cases intro!: ntr.intro gtr1-s trss.intro)
  — That can be appended to the initial normalized path
  from trcl-rev-cons[OF IHP'(1) this] have ?case by blast
} moreover {
— Last step was no procedure call
  fix q a
  assume CASE: e=LBase a ∨ e=LSpawn q ∨ e=LRet
  — Then it is a same-level or returning path
  with TSTEP(3) obtain u r r' where SLR: s=u# r s'=r'@[r length r' ≤ 1
(((u), ce), e), (r', ce')) ∈ trcl (trss fg) by (force elim!: trss.cases intro!: trss.intro)
  — That can be appended to the normalized path using the []
  {} # r' @ ?r# + ?ce')} ∈ trcl (ntr fg) - lemma
  from ntr-sl-move-left[OF - SLR(4) IHP(5) SLR(3)] IHP'(1) TSTEP(1)
  SLR(1) obtain ww' where ((# [entry fg p] #), ww', {# r' @ r#} + ce') ∈ trcl (ntr fg) by auto
  with SLR(2) TSTEP(2) have ?case by auto
} ultimately show ?case by (cases e, auto)
qed

As the main result of this section we get: A configuration is reachable if and only if it is also reachable via a normalized path:
theorem (in eflowgraph) ntr-repr:
\[(\exists w. \{ \#entry fg (main fg)\} \cup \{ w \}, c) \in trcl (tr fg) \iff (\exists w. \{ \#entry fg (main fg)\} \cup \{ w \}, c) \in trcl (ntr fg)\]
by (auto simp add: initialproc-def intro: normalize ntr-is-tr)

8.4 Properties of normalized path

Like a usual path, also a macrostep modifies one thread, spawns some threads and preserves the state of all the other threads. The spawned threads do not make any steps, thus they stay in their initial configurations.

lemma ntrs-c-cases-s [cases set]: \[
\{\langle (s, c), w, (s', c')\rangle \in ntr fg; \quad
\langle (s, c), w, (s', c')\rangle \in trcl (ntr fg)\}\] !csp. \[\langle c' = csp + c; \langle s, s: #csp \Rightarrow \exists p u v. s = [entry fg p] \land (u, Spawn p, v) \in edges fg \land initialproc fg p\rangle\]\[\Rightarrow P\] 
by (auto dest!: ntrs-is-trss-s elim!: trss-c-cases)

lemma ntrs-c-cases [cases set]: \[
\{\langle (s, c), w, (s', c')\rangle \in ntr fg; \quad
\langle (s, c), w, (s', c')\rangle \in trcl (ntr fg)\}\] !csp. \[\langle c' = csp + c; \langle s, s: #csp \Rightarrow \exists p u v. s = [entry fg p] \land (u, Spawn p, v) \in edges fg \land initialproc fg p\rangle\]\[\Rightarrow P\] 
by (auto dest!: ntrs-is-trss-s elim!: trss-c-cases)

8.4.1 Validity

Like usual paths, also normalized paths preserve validity of the configurations.

lemmas (in flowgraph) ntr-valid-preserve-s = trss-valid-preserve[OF ntrs-is-trss-s]
lemmas (in flowgraph) ntr-valid-preserve-s = tr-valid-preserve[OF ntr-is-tr-s]
lemmas (in flowgraph) ntr-valid-preserve = tr-valid-preserve[OF ntr-is-tr]
lemmas (in flowgraph) ntrp-valid-preserve-s [cases set]: [\langle (s, c), w, (s', c')\rangle \in ntrp fg] !csp. \[\langle c' = csp + c; \langle s, s: #csp \Rightarrow \exists p u v. s = [entry fg p] \land (u, p, v) \in ntrp edges (fg, p) \land initialproc fg p\rangle\]\[\Rightarrow P\] 
by (auto dest!: ntrs-is-trss-s elim!: trss-c-cases)

lemma (in flowgraph) ntrp-valid-preserve: assumes A: [\langle (s, c), w, (s', c')\rangle \in trcl (ntrp fg)] !csp. \[\langle c' = csp + c; \langle s, s: #csp \Rightarrow \exists p u v. s = [entry fg p] \land (u, p, v) \in trcl (ntrp edges (fg, p) \land initialproc fg p)\rangle\]\[\Rightarrow P\] 
by assumption
8.4.2 Monitors

The following defines the set of monitors used by a normalized path and shows its basic properties:

**definition**

\[ \text{mon-ww } fg \, ww \equiv \text{foldl} \left( \text{op } \cup \right) \left\{ \right\} \left( \text{map} \left( \text{mon-w } fg \right) \, ww \right) \]

**definition**

\[ \text{mon-loc } fg \, ww \equiv \text{mon-ww } fg \left( \text{map } \text{le-rem-s } \left( \text{loc } ww \right) \right) \]

**definition**

\[ \text{mon-env } fg \, ww \equiv \text{mon-ww } fg \left( \text{map } \text{le-rem-s } \left( \text{env } ww \right) \right) \]

**lemma** **mon-ww-empty**

\[ \text{mon-ww-empty } [\text{simp}]: \text{mon-ww } fg \, [] = \{\} \]

**by (unfold mon-ww-def, auto)**

**lemma** **mon-ww-uncons**

\[ \text{mon-ww-uncons } [\text{simp}]: \text{mon-ww } fg \, (ee \# \text{ww}) = \text{mon-w } fg \, ee \cup \text{mon-ww } fg \, \text{ww} \]

**by (unfold mon-ww-def, auto simp add: foldl-un-empty-eq[of mon-w fg ee])**

**lemma** **mon-env-empty**

\[ \text{mon-env-empty } [\text{simp}]: \text{mon-env } fg \, [] = \{\} \]

**by (unfold mon-env-def)**

**lemma** **mon-env-single**

\[ \text{mon-env } fg \, [e] = (\text{case } e \, \text{of } \text{LOC a } \Rightarrow \{\} \mid \text{ENV a } \Rightarrow \text{mon-w } fg \, a) \]

**by (unfold mon-env-def) (auto split: el-step.split)**

**lemma** **mon-env-uncons**

\[ \text{mon-env-uncons } [\text{simp}]: \text{mon-env } fg \, (e \# \text{w}) = (\text{case } e \, \text{of } \text{LOC a } \Rightarrow \{\} \mid \text{ENV a } \Rightarrow \text{mon-w } fg \, a) \cup \text{mon-env } fg \, w \]

**by (unfold mon-env-def) (auto split: el-step.split)**

**lemma** **mon-env-unconc**

\[ \text{mon-env-unconc}: \text{mon-env } fg \, (\text{ww1} \circ\circ \text{ww2}) = \text{mon-ww } fg \, \text{ww1} \cup \text{mon-ww } fg \, \text{ww2} \]

**by (induct \text{ww1}) auto**

**lemma** **mon-loc-empty**

\[ \text{mon-loc-empty } [\text{simp}]: \text{mon-loc } fg \, [] = \{\} \]

**by (unfold mon-loc-def) auto**

**lemma** **mon-loc-single**

\[ \text{mon-loc-single } [\text{simp}]: \text{mon-loc } fg \, [e] = (\text{case } e \, \text{of } \text{ENV a } \Rightarrow \{\} \mid \text{LOC a } \Rightarrow \text{mon-w } fg \, a) \]

**by (unfold mon-loc-def) (auto split: el-step.split)**

**lemma** **mon-loc-uncons**

\[ \text{mon-loc-uncons } [\text{simp}]: \text{mon-loc } fg \, (e \# \text{w}) = (\text{case } e \, \text{of } \text{ENV a } \Rightarrow \{\} \mid \text{LOC a } \Rightarrow \text{mon-w } fg \, a) \cup \text{mon-loc } fg \, w \]

**by (unfold mon-loc-def) (auto split: el-step.split)**

**lemma** **mon-loc-unconc**

\[ \text{mon-loc-unconc}: \text{mon-loc } fg \, (\text{ww1} \circ\circ \text{ww2}) = \text{mon-loc } fg \, \text{ww1} \cup \text{mon-loc } fg \, \text{ww2} \]

**by (unfold mon-loc-def) (auto simp add: mon-ww-unconc)**
lemma mon-ww-of-foldl[simp]: mon-w fg (foldl (op @) [] ww) = mon-ww fg ww
apply (induct ww)
apply (unfold mon-ww-def)
apply simp
apply simp
apply (subst foldl-conc-empty-eq, subst foldl-un-empty-eq)
apply (simp add: mon-w-unconc)
done

lemma mon-ww-ileq: w ⪯ w' ⇒ mon-ww fg w ⊆ mon-ww fg w'
by (induct rule: less-eq-list-induct) auto

lemma mon-ww-cil: w ∈ w1 ⊗ α w2 ⇒ mon-ww fg w = mon-ww fg w1 ∪ mon-ww fg w2
by (induct rule: cil-set-induct-fix α) auto

lemma mon-env-cil: w ∈ w1 ⊗ α w2 ⇒ mon-env fg w = mon-env fg w1 ∪ mon-env fg w2
by (induct rule: cil-set-induct-fix α) auto

lemma mon-ww-of-le-rem: mon-ww fg (map le-rem-s w) = mon-loc fg w ∪ mon-env fg w
by (induct w) (auto split: el-step.split)

lemma mon-env-map-env[simp]: mon-env fg (map ENV w) = mon-ww fg w
by (unfold mon-env-def) simp

lemma mon-env-map-loc[simp]: mon-env fg (map LOC w) = {}
by (induct w) auto

— As monitors are syntactically bound to procedures, and each macrostep starts
with a non-returning call, the set of monitors allocated during the execution of a
normalized path is monotonically increasing
lemma (in flowgraph) ntrs-mon-increasing-s: ((s,c),e,(s',c'))∈ntrs fg
⇒ mon-s fg s ⊆ mon-s fg s' ∧ mon-c fg c = mon-c fg c'
apply (erule ntrs.cases)
apply (auto simp add: trss-c-no-mon)
apply (subgoal_tac mon-n fg u = mon-n fg u')
apply (simp)
apply (auto elim!: trss.cases dest!: mon-n-same-proc edges-part)
done

lemma (in flowgraph) ntr-mon-increasing-s:
  \((s,c),(s',c')\)\(\in\)ntr fg \(\Rightarrow\) mon-c fg c \(\subseteq\) mon-c fg c'
by (erule gtrE) (auto dest: ntrs-mon-increasing-s simp add: mon-c-unconc)

lemma (in flowgraph) ntrp-mon-increasing-s:
  \((s,c),(s',c')\)\(\in\)ntrp fg \(\Rightarrow\) mon-s fg s \(\subseteq\) mon-s fg s' \(\land\) mon-c fg c \(\subseteq\) mon-c fg c'
apply (erule gtrp.cases)
apply (auto dest: ntrs-mon-increasing-s simp add: mon-c-unconc)
apply (erule ntrs-c-cases-s)
apply auto

proof
  --
  fix c c' s' x csp
  assume {#s#} + c' = csp + ({#s#} + c)
  with union-left-cancel[of {#s#} c' csp+e] have c' = csp+e by (simp add: union-ac)
  moreover assume \(x\) \(\in\) mon-c fg c \(\notin\) mon-c fg c'
  ultimately have False by (auto simp add: mon-c-unconc)
  thus \(x\) \(\in\) mon-s fg s' ..
qed

lemma (in flowgraph) ntrp-mon-increasing:
  \(((s,c),e,(s',c'))\)\(\in\)trcl (ntrp fg)
  \(\Rightarrow\) mon-s fg s \(\subseteq\) mon-s fg s' \(\land\) mon-c fg c \(\subseteq\) mon-c fg c'
by (induct rule: trcl-rev-pair-induct) (auto dest!: ntrs-mon-increasing-s)

8.4.3 Modifying the context

lemmas (in flowgraph) ntrs-c-no-mon-s = trss-c-no-mon[OF ntrs-is-trss-s]
lemmas (in flowgraph) ntrs-c-no-mon = trss-c-no-mon[OF ntrs-is-trss]

Also like a usual path, a normalized step must not use any monitors that
are allocated by other threads

lemmas (in flowgraph) ntrs-mon-e-no-ctx = trss-mon-w-no-ctx[OF ntrs-is-trss-s]
lemma (in flowgraph) ntrs-mon-e-no-ctx:
  assumes A: \(((s,c),w,(s',c'))\)\(\in\)trcl (ntrs fg)
  shows mon-ww fg w \(\cap\) mon-c fg c = {}
using trss-mon-w-no-ctx[OF ntrs-is-trss[OF A]] by simp

lemma (in flowgraph) ntrp-mon-env-e-no-ctx:
  \(((s,c),ENV e,(s',c'))\)\(\in\)ntrp fg \(\Rightarrow\) mon-w fg e \(\cap\) mon-s fg s = {}
by (auto elim!: gtrp.cases dest!: ntrs-mon-e-no-ctx simp add: mon-c-unconc)
lemma (in flowgraph) ntrp-mon-loc-e-no-ctx:
  \(((s,c),LOC e,(s',c'))\)\(\in\)ntrp fg \(\Rightarrow\) mon-w fg e \(\cap\) mon-e fg c = {}
by (auto elim!: gtrp.cases dest!: ntrs-mon-e-no-ctx)
The next lemmas are rules how to add or remove threads while preserving the executability of a path:

**Lemma (in flowgraph) ntrs-modify-context-s:**
- **Assumes:** $A = ((s, c, e), (s', c')) \in \text{ntrs}\ fg$
- **And:** $B = \text{mon-w}\ fg\ ee \cap \text{mon-c}\ fg\ cn = \{}$
- **Shows:** $\exists\ c'. c' = csp + c \land \text{mon-c}\ fg\ esp = \{} \land ((s, cn), ee, (s', csp + cn)) \in \text{ntrs}\ fg$
- **Proof:**
  - From $A$ obtain $p r u u' v w$ where $S: s = u # r ee = \text{LCall}\ p # w s' = v # u' # r$
  - $(u # r, c), \text{LCall}\ p, (\text{entry}\ fg\ p # u' # r, c) \in \text{trss}\ fg$ $\Rightarrow (\text{entry}\ fg\ p), c, w, ([u], [v], [c']) \in \text{trcl}\ (\text{trss}\ fg)$ by (blast elim!: ntrs.cases[simplified])
  - With $\text{trss-modify-context-s}[\text{OF}\ S(4)] B$ have $(u # r, cn), \text{LCall}\ p, (\text{entry}\ fg\ p # u' # r, cn) \in \text{trss}\ fg$ by auto
  - Moreover from $S$ $\text{trss-modify-context}[\text{OF}\ S(5)] B$ obtain $csp$ where $c' = csp + c$
  - **Mon-c\ fg\ esp = \{} \land ((s, cn), ee, (s', csp + cn)) \in \text{trcl}\ (\text{trss}\ fg)$ by auto
  - Ultimately show $\text{thesis using } S$ by (auto intro!: ntrs-step)
- **Qed**

**Lemma (in flowgraph) ntrs-modify-context-rule-format:**
- $[[(s, c), (s', c') \in \text{trcl}\ (\text{ntrs}\ fg)]]$
- $\Rightarrow \forall\ cn. \text{mon-ww}\ fg\ w\ \cap\ \text{mon-c}\ fg\ cn = \{}$
- $\Rightarrow (\exists\ csp. c' = csp + c \land \text{mon-c}\ fg\ esp = \{} \land ((s, cn), w, (s', csp + cn)) \in \text{trcl}\ (\text{ntrs}\ fg))$
- **Proof:**
  - **(induct rule: trcl-pair-induct)**
  - **Case empty** thus case by simp
  - **Next**
    - **Case (cons s c e sh ch w s' c') note IHP=this show ?case**
    - **Proof:** (intro allI simp)
      - Fix $cn$
      - Assume $\text{MON}: \text{mon-ww}\ fg\ (c \not\# w) \cap \text{mon-c}\ fg\ cn = \{}$
      - From ntrs-modify-context-s[\text{OF}\ IHP(1)] $\text{MON}$ obtain $csph$ where $S1: c = csp + c$ mon-c\ fg\ esp = \{} $(s, cn), e, sh, csph + cn) \in \text{ntrs}\ fg$ by auto
      - With $\text{MON}$ have $\text{mon-ww}\ fg\ w\ \cap\ \text{mon-c}\ fg\ (csph + cn) = \{}$ by (auto simp add: mon-c-unc onc)
      - With $\text{IHP(3)}[\text{rule-format}]$ obtain $csp$ where $S2: c' = csp + ch$ mon-c\ fg\ esp = \{} $(sh, csp + cn), w, (s', csp + (csp + cn))) \in \text{trcl}\ (\text{ntrs}\ fg)$ by blast
      - From $S1$ $S2$ have $c' = c + ch$ mon-c\ fg\ esp = \{} $(s, cn), e\# w, (s', csp + (csp + cn)) \in \text{trcl}\ (\text{ntrs}\ fg)$ by (auto simp add: union-assoc mon-c-unc onc)
      - Thus $\exists\ csp. c' = csp + c \land \text{mon-c}\ fg\ esp = \{} \land ((s, cn), e \# w, s', csp + cn) \in \text{trcl}\ (\text{ntrs}\ fg)$ by blast

64
proof

lemma ntrs-xchange-context-s:
  assumes A: \(((s,c), ee,(s',csp+c))\) \in ntrs fg
  and B: mon-c fg cn \subseteq mon-c fg c
  shows \(((s, cn), ee,(s',csp+cn))\) \in ntrs fg
proof –
  obtain \(p r u u' v w\) where \(S: s = u \# r\ ee = LCall p \# w s' = v \# u' \# r\) \((u \# r,c), LCall p, (entry fg p \# u' \# r,c)\) \in trss fg 
  \(((\text{entry fg p}, c), w, ([v],csp+c))\) \in trcl (trss fg)
proof –
  case goal1 moreover
  from ntrs.cases[OF A, simplified] obtain ce ce' \(p r u u' v w\) where \(s = u \# r\ c = ce ee = LCall p \# w s' = v \# u' \# r\) csp + ce = ce' \((u \# r, ce), LCall p, entry fg p \# u' \# r, ce)\) \in trss fg
  \(((\text{entry fg p}, ce), w, ([v], ce')\) \in trcl (trss fg) .
  hence \(s = u \# r\ ee = LCall p \# w s' = v \# u' \# r\) \((u \# r,c), LCall p, (entry fg p \# u' \# r,c)\) \in trss fg
  \(((\text{entry fg p}, c), w, ([v],csp+c))\) \in trcl (trss fg) by auto
  ultimately show ?thesis .
  qed

qed

lemma ntrs-replace-context-s:
  assumes A: \(((s,c+cr), ee,(s',c'+cr))\) \in ntrs fg
  and B: mon-c fg crn \subseteq mon-c fg cr
  shows \(((s,c+crn), ee,(s',c'+crn))\) \in ntrs fg
proof –
  from ntrs-cases-s[OF A] obtain csp where G: \(c'+cr = csp+(c+cr)\) . hence
  F: \(c'=csp+c\) by (auto simp add: union-assoc[symmetric])
  from B have MON: mon-c fg \((c+crn)\) \in mon-c fg \((c+cr)\) by (auto simp add: mon-c-unconc)
  from ntrs-xchange-context-s[OF - MON] A G have \(((s,c+crn), ee,(s',csp+(c+crn))\) \in ntrs fg by auto
  with F show ?thesis by (simp add: union-assoc)
qed

lemma (in flowgraph) ntrs-xchange-context: \(!!s c c' cn. [\)
  \((s,c),ww,(s',c')\) \in trcl (ntrs fg);
  mon-c fg cn \subseteq mon-c fg c
\] \Rightarrow \exists csp.
  \(c'=csp+c \wedge \((s,cn),ww,(s',csp+cn)\) \in trcl (ntrs fg)

proof (induct \(ww\)
  case Nil note CASE=this
  thus ?case by (auto intro! : \(exI[of - \{#\}]\))
next
  case (Cons ee \(ww\)) note IHP=this
  then obtain sh ch where SPLIT: \(((s,c),ee,(sh, ch))\) \in ntrs fg 
  \(((sh, ch),ww,(s',c')\) \in trcl

65
(ntrs fg) by (fast dest: trcl-uncons)
from ntrs-cases-s[of SPLIT(1)] obtain csph where CHFMT: ch=csp+ch 
!!s. s:#csph ⇒ ∃p u v. s=[entry fg p] ∧ (u, Spawn p, v) ∈ edges fg ∧ initialproc
fg p by blast
with ntrs-xchange-context-s SPLIT(1) IHP(3) have ((s,cn),ce,(sh,csph+cn))∈ntrs
fg by blast
also
from c-of-initial-no-mon CHFMT(2) have CSPH-NO-MON: mon-c fg csph = 
{} by auto
with IHP(3) CHFMT have 1: mon-c fg (csph+cn) ⊆ mon-c fg ch by (auto simp add: mon-c-unconc)
from IHP(1)[OF SPLIT(2) this] obtain csp where C'FMT: c'=csp+ch and
SND: ((sh,csph+cn),ww,(s',csp+(csph+cn)))∈trcl (ntrs fg) by blast
note SND
finally have ((s, cn), ee # ww, s', (csp + csph) + cn) ∈ trcl (ntrs fg) by (simp add: union-assoc)
moreover from CHFMT(1) C'FMT have c'=(csp+csph)+c by (simp add: union-assoc)
ultimately show ?case by blast
qed

lemma (in flowgraph) ntrs-replace-context:
  assumes A: ((s,c+cr),ww,(s',c'+cr))∈trcl (ntrs fg)
  and B: mon-c fg crn ⊆ mon-c fg cr
  shows ((s,c+crn),ww,(s',c'+crn))∈trcl (ntrs fg)
proof –
  from ntrs-cases[of A] obtain csp where G: c'+cr = csp+(c+cr) . hence
  F: c'=csp+c by (auto simp add: union-assoc[symmetric])
  from B have MON: mon-c fg (c+crn) ⊆ mon-c fg (c+cr) by (auto simp add:
  mon-c-unconc)
  from ntrs-xchange-context[of A MON] G have ((s,c+crn),ww,(s',csp+(c+crn)))∈trcl
  (ntrs fg) by auto
  with F show ?thesis by (simp add: union-assoc)
qed

lemma (in flowgraph) ntr-add-context-s:
  assumes A: (e,e',e')∈ntr fg
  and B: mon-w fg e ∩ mon-c fg cn = {}
  shows (c+cn,e,e'+cn)∈ntr fg
proof –
  from gtrE[of A] obtain s ce s' ce' where NTRS: c = {#s#} + ce e' = 
  {#s'#} + ce' ((s, ce), e, s', ce') ∈ ntrs fg .
  from ntrs-mon-e-no-ctx[of NTRS(3)] B have M: mon-w fg e ∩ (mon-c fg
  (ce+cn)) = {} by (auto simp add: mon-c-unconc)
  from ntrs-modify-context-s[of NTRS(3) M] have ((s,ce+cn),e,(s',ce'+cn))∈ntrs
  fg by (auto simp add: union-assoc)
  with NTRS show ?thesis by (auto simp add: union-assoc intro: gtr1-s)
qed
lemma (in flowgraph) ntr-add-context:
\[(c,w,c')\in trcl (ntr fg)\); mon-ww fg w \cap mon-c fg cn = {}\]
\[\implies (c+cn,w,c'+cn)\in trcl (ntr fg)\]
by (induct rule: trcl.induct) (simp, force dest: ntr-add-context-s)

lemma (in flowgraph) ntr-add-context-s:
assumes A: \[((s,c),e,(s',c'))\in ntrs fg\]
and B: mon-w fg e \cap mon-c fg cn = {}
shows \[((s,c+cn),e,(s',c'+cn))\in ntrs fg\]
(force simp add: mon-e-unconc union-ac)

lemma (in flowgraph) ntrp-add-context-s:
\[\[(s,c),e,(s',c')\in ntrp fg; mon-w fg (le-rem-s e) \cap mon-c fg cn = {}\]\]
\[\implies ((s,c+cn),(e,(s',c'+cn))\in ntrp fg\]
apply (erule gtrp_cases)
apply (auto dest: ntrp-add-context-s intro!: gtrpintros)
apply (simp only: unionassoc)
apply (rule gtrp-env)
apply (simp only: unionassoc[symmetric])
apply (rule ntrp-add-context-s)
apply assumption+
done

lemma (in flowgraph) ntrp-add-context:
\[(s,w,(s',c'))\in trcl (ntrp fg)\]
\[mon-ww fg (map le-rem-s w) \cap mon-c fg cn = {}\]
\[\implies ((s,c+cn),w,(s',c'+cn))\in trcl (ntrp fg)\]
by (induct rule: trcl-pair-induct) (simp, force dest: ntrp-add-context-s)

8.4.4 Altering the local stack

lemma ntrs-stack-comp-s:
assumes A: \[((s,c),ee,(s',c'))\in ntrs fg\]
shows \[((s@r,c),ee,(s'@r,c'))\in ntrs fg\]
using A
by (auto dest: trss-stack-comp trss-stack-comp-s elim!: ntrs_cases intro!: ntrs-step[simplified])

lemma ntrs-stack-comp: \[((s,c),ww,(s',c'))\in trcl (ntrs fg)\]
\[\implies ((s@r,c),ww,(s'@r,c'))\in trcl (ntrs fg)\]
by (induct rule: trcl-pair-induct) (auto intro!: trcl_cons[OF ntrs-stack-comp-s])

lemma (in flowgraph) ntrp-stack-comp-s:
assumes A: \[((s,c),ee,(s',c'))\in ntrp fg\]
and B: mon-s fg r \cap mon-env fg [ee] = {}
shows \[((s@r,c),ee,(s'@r,c'))\in ntrp fg\]
using A

67
proof (cases rule: gtrp.cases)
  case gtrp-loc then obtain e where CASE: ee=LOC e ((s,c),e,(s',c'))∈ntrs fg
  by auto
  hence ((s@r),e,(s'@r,c'))∈ntrs fg by (blast dest: ntrs-stack-comp-s)
  with CASE(1) show ?thesis by (auto intro: gtrp.gtrp-loc)
next
  case gtrp-env then obtain sm ce sm’ ce’ e where CASE: s’=s c={#sm#}+ce
c’={#sm’#}+ce’ ee=ENV e ((sm,{#s#}+ce),e,(sm’,{#s’#}+ce’))∈ntrs fg by auto
  from ntrs-modify-context-s[OF CASE(5), where cn={#s'r#}+ce] ntrs-mon-e-no-ctx[OF CASE(4)] obtain csp where
    ADD: {#s#}+ce’ = csp + ({#s#}+ce) mon-e fg csp = {} ((sm, {#s
    @ r#}+ce), e, sm’, csp + ({#s @ r#} + ce)) ∈ ntrs fg by (auto simp add: mon-c-unconc mon-s-unconc)
  moreover from ADD(1) have {#s#}+ce’={#s#}+(csp+ce) by (simp add: union-ac) hence ce’=csp+ce by simp
  ultimately have (sm, {#s @ r#} + ce), e, sm’, (m # s @ r#) + ce’) ∈ ntrs fg by (simp add: union-ac)
  with CASE(1,2,3,4) show ?thesis by (auto intro: gtrp.gtrp-env)
qed

lemma (in flowgraph) ntrp-stack-comp:
  [ ((s,c),ww,(s’,c’))∈trcl (ntrp fg); mon-s fg r ∩ mon-env fg ww = {} ]
  ⟹ ((s@r),c),ww,(s'@r,c')∈trcl (ntrp fg)
  by (induct rule: trcl-pair-induct) (auto intro!: trcl.cons[OF ntrp-stack-comp-s])

lemma ntrp-stack-top-decomp-s:
  assumes A: ((u#r),e,e,(s',c'))∈ntrs fg
  and EX: !!v u’. [ s'=v#u'#r;
    (vu],[u,c),ee,([v,u'],c')∈ntrs fg;
    (u,Call p,u')∈edges fg ] ⟹ P
  shows P
  using A
  proof (cases rule: ntrp.cases)
    case ntrp-step then obtain u' v p w where CASE: ee=LCall p#w s'=v#u'#r
      ((u#r),c),LCall p,entry fg p#u'#r,c))∈trss fg ((entry fg p,c),w,([v,c'])∈trcl (trss fg)
      by (simp)
    from trss-stack-decomp-s[where s=[u], simplified, OF CASE(3)] have SDC:
      (((u],c), LCcall p, (entry fg p, u', c)) ∈ trss fg by auto
    with CASE(1,4) have ((u],c),ee,([v,u'],c')∈ntrs fg by (auto elim!: ntrs.ntrp-step)
    moreover from SDC have (u,Call p,u')∈edges fg by (auto elim!: trss.cases)
    ultimately show ?thesis using CASE(2) by (blast intro!: EX)
  qed

lemma ntrp-stack-decomp-s:
  assumes A: ((u#s'r),e,e,(s',c'))∈ntrs fg
  and EX: !!v u'. [ 
\[ s' = v\#u'\#s\#r; \\
((u\#s,c), ee,(v\#u'\#s,c')) \in \text{ntrp fg}; \\
(u, \text{Call } p,u') \in \text{edges fg} \]

shows \( P \)

apply (rule \text{ntrs-stack-top-decomp-s[OF A]})
apply (rule \text{EX})
apply (auto dest: \text{ntrs-stack-comp-s})
done

\text{lemma ntrs-stack-decomp:} \forall u \; s \; r \; c. \; \exists \; P \]
\[
((u\#s,r,c), ww,(s',c')) \in \text{trcl} (\text{ntrp fg}); \\
!!v \; rr. \; [s'=v\#rr\#r; ((u\#s,c), ee,(v\#rr,c')) \in \text{trcl} (\text{ntrp fg})] \implies P 
\]

\text{proof} (induct \text{ww})
\text{case Nil thus \text{thesis by fastforce}}
next
\text{case} (\text{Cons } e \; w) \; \text{from} \; \text{Cons.prems show} \; \text{case proof} (\text{cases rule: trcl-pair-unconsE})
\text{case} (\text{split } sh \; ch)
from \text{ntrs-stack-decomp-s[OF \; \text{split}(1)]} \; \text{obtain} \; vh \; uh \; p \; \text{where} \; F: \; sh = vh\#uh\#s\#r
\[
((u\#s, c), e, vh\#uh\#s, ch) \in \text{ntrp fg} \; (u, \text{Call } p, u, vh, uh) \in \text{edges fg} \text{ by blast} \\
\text{from} \; \text{F(1) \; \text{split}(2) \; \text{Cons.hyps[of vh uh s r ch]} \; \text{obtain} \; v' \; rr \; \text{where}} \; S: \\
s' = v'\#rr\#r; ((vh\#uh\#s,ch),(v'\#rr,c')) \in \text{trcl} (\text{ntrp fg}) \; \text{by auto} \\
\text{from} \; \text{trcl.cons[OF \; \text{F(2)} \; \text{S(2)}]} \; \text{S(1) \; \text{Cons.prems(2) show} \; \text{thesis by blast}}
\text{qed}
\text{qed}

\text{lemma ntrp-stack-decomp-s:}
\text{assumes} \; A: ((u\#s,r,c), ee,(s',c')) \in \text{ntrp fg} \\
\text{and} \; \text{EX:} \; \forall v \; rr. \; [s'=v\#rr\#r; ((u\#s,c), ee,(v\#rr,c')) \in \text{ntrp fg}] \implies P 
\text{shows} \; P 
\text{using} \; A
\text{proof} (\text{cases rule: gtrp.cases})
\text{case gtrp-loc} \; \text{thus \text{thesis using} \; \text{EX by} (\text{force elim!: ntrsp-stack-decomp-s intro!: gtrp.intros})}
next
\text{case gtrp-env} \; \text{then obtain} \; e \; ss \; ss' \; ce \; ce' \; \text{where} \; S: \; ee = \text{ENV} \; e \; s' = u\#s\#r \\
c = \{\#s\#\} + ce \; c' = \{\#s'\#\} + ce' \; (ss, ce + \{\#u\#s\#r\#\}), e,(ss', ce' + \{\#u\#s\#r\#\}) \in \text{ntrp fg} \text{ by (auto simp add: union-ac)} \\
\text{from} \; \text{ntrs-replace-context-s[OF \; \text{S(5)}, \text{where crn} = \{\#u\#s\#\}] \; \text{have} \; ((ss, \{\#u \; s\#\} + ce), e, ss', \{\#u \; s\#\} + ce') \in \text{ntrp fg} \text{ by (auto simp add: mon-s-unconc union-ac)} \\
\text{with} \; S \text{ show} \; P \text{ by (rule-tac \; \text{EX}) (auto intro: gtrp.gtrp-env)}
\text{qed}

\text{lemma ntrp-stack-decomp:} \forall u \; s \; r \; c. \; \exists \; P \\
((u\#s,r,c), ww,(s',c')) \in \text{trcl} (\text{ntrp fg}); \\
!!v \; rr. \; [s'=v\#rr\#r; ((u\#s,c), ee,(v\#rr,c')) \in \text{trcl} (\text{ntrp fg})] \implies P 
\]

\text{qed}
proof (induct ww)
case Nil thus \textit{case} by \textit{fastforce}
next
case (Cons e w) from \textit{Cons.prems show} \textit{case proof} (cases rule: trcl-pair-unconsE)
case (split sh ch)
  from ntrp-stack-decomp-s[of split(1)] obtain vh rrh where \( F \) \( sh = vh \# rrh \# ch \) in ntrp fg by blast
  from F(1) split(2) Cons.hyps[of vh rrh r ch] obtain v' rr where \( S = s' = v' \# rr \# ch \)\( (vh \# rrh, ch), v, (v' \# rr, c') \)\( \in \)\( trcl \)\( (ntrp fg) \) by auto
  from trcl.cons[OF F(2) S(2)] S(1) Cons.prems(2) show \textit{thesis} by blast
qed
qed

8.5 Relation to monitor consistent interleaving

In this section, we describe the relation of the consistent interleaving operator (cf. Section 2) and the macrostep-semantics.

8.5.1 Abstraction function for normalized paths

We first need to define an abstraction function that maps a macrostep on a pair of entered and passed monitors, as required by the \( \otimes_{\alpha} \)-operator:

A step on a normalized paths enters the monitors of the first called procedure and passes the monitors that occur in the following same-level path.

definition \( \alpha_{n fg e} = \) \( \begin{cases} \{\}, \{\} & \text{if} \ e = [] \text{ then} \{\}, \{\} \\ (\text{mon-e} \ fg \ (\text{hd} \ e), \text{mon-w} \ fg \ (\text{tl} \ e)) & \text{else} \end{cases} \)

lemma \( \alpha_{n-simps} \) [simp]:
  \( \alpha_{n fg} [] = \{\}, \{\} \)
  \( \alpha_{n fg} (e \# w) = (\text{mon-e} \ fg \ e, \text{mon-w} \ fg \ w) \)
  by (unfold \( \alpha_{n-def} \), auto)

— We also need an abstraction function for normalized \textit{loc/env-paths}

definition \( \alpha_{nl fg e} = \alpha_{n fg} \circ \text{le-rem-s} \ e \)

lemma \( \alpha_{nl-def} \) [simp]: \( \alpha_{nl fg} \circ \text{ENV} = \alpha_{n fg} \)
  by (rule eq-reflection[OF ext]) (auto simp add: \( \alpha_{nl-def} \))

— These are some ad-hoc simplifications, with the aim at converting \( \alpha_{nl} \) back to \( \alpha_{n} \)

lemma \( \alpha_{nl-simps} \) [simp]:
  \( \alpha_{nl fg} \ (\text{ENV} \ x) = \alpha_{n fg} \ x \)
  \( \alpha_{nl fg} \ (\text{LOC} \ x) = \alpha_{n fg} \ x \)
  by (unfold \( \alpha_{nl-def} \), auto)

lemma \( \alpha_{nl-simps1} \) [simp]:
  \( \alpha_{nl fg} \circ \text{ENV} = \alpha_{n fg} \)
\[(\alpha n fg) \circ LOC = \alpha n fg\]

by \((\text{unfold} \ \alpha n\ \text{def'} \ \text{comp-def})\) (simp-all)

**Lemma** \(\alpha n\-\alpha nl\): \((\alpha n fg) \circ le-rem-s = \alpha nl fg\)

**Unfolding** \(\alpha n\-fst-snd\) \([\text{simp}]:\) \(\text{fst} (\alpha n fg w) \cup \text{snd} (\alpha n fg w) = \text{mon-w fg w}\)

by \((\text{induct w})\) auto

**Lemma** \(\text{mon-pl-of-} \alpha nl\): \(\text{mon-pl} (\text{map} (\alpha nl fg) w) = \text{mon-loc fg w} \cup \text{mon-env fg w}\)

by \((\text{induct w})\) (auto split: \(\text{el-step}\). split)

We now derive specialized introduction lemmas for \(\otimes \alpha n fg\)

**Lemma** \(cil\-\alpha n\-cons-helper\):
\(\text{mon-pl} (\text{map} (\alpha n fg) wb) = \text{mon-ww fg wb}\)

\(\text{by (simp add:} \ \alpha n\-\alpha nl \ \text{cil}{}\-\alpha n\-\text{cons-helper}[\text{symmetric}]\))

**Lemma** \(cil\-\alpha nl\-cons-helper\):
\(\text{mon-pl}(\text{map}(\alpha nl fg) wb) = \text{mon-ww fg } (\text{map le-rem-s wb})\)

\(\text{by (simp add:} \ \alpha n\-\alpha nl \ \text{cil}{}\-\alpha n\-\text{cons-helper}\))

**Lemma** \(cil\-\alpha n\-cons1\): \(w \in wa \otimes \alpha n fg \ \text{wb}; \ \text{fst} (\alpha n fg e) \cap \text{mon-ww fg wb} = \{\}\)

\(\Rightarrow \ e \# w \in e \# wa \otimes \alpha n fg \ \text{wb}\)

\(\text{by (rule cil-cons1)}\)

**Lemma** \(cil\-\alpha n\-cons2\): \(w \in wa \otimes \alpha n fg \ \text{wb}; \ \text{fst} (\alpha n fg e) \cap \text{mon-ww fg wa} = \{\}\)

\(\Rightarrow \ e \# w \in wa \otimes \alpha n fg e \# wb\)

**8.5.2 Monitors**

**Lemma** \(\text{in flowgraph})\ \text{ntrs-mon-s}\):
\(\text{assumes} A: ((s,c),e,(\text{\prime},c)) \in \text{ntrs fg}\)

\(\text{shows} \ \text{mon-s fg s' = mon-s fg s \cup \text{fst} (\alpha n fg e)}\)

\(\text{proof} - \)

\(\text{from} A \ \text{obtain} u r p u' w v \ \text{where DET:} s = u \# r e = \text{LCall p} \# w ((u \# r, c), \text{LCall p, (entry fg p \# u' \# r, c)) \in trss fg (([entry fg p], c), w, ([v], c'))} \in \text{trcl} (\text{trss fg}) s' = v \# u' \# r\)

71
by (blast elim!: ntrs_cases[simplified])

hence mon-n fg u = mon-n fg u' by (auto elim!: trss_cases dest: mon-n-same-proc edges-part)

with trss-bot-proc-const[where s=[], and s'=[]], simplified, OF DET(4)] DET(1,2,5)
show thesis by (auto simp add: mon-n-def an-def)

qed

corollary (in flowgraph) ntrs-called-mon:
assumes A: ((s,c),e,(s',c'))\in ntrs fg
shows fst (\alpha_n fg e) \subseteq mon-s fg s'
using ntrs-mon-s[OF A] by auto

lemma (in flowgraph) ntrp-mon-s:
assumes A: ((s,c),e,(s',c'))\in ntrp fg
shows mon-c fg \{{\#s\#}\} + c' = mon-c fg (\{{\#s\#}\} + e) \cup fst (\alpha_nl fg e)
using ntr-mon-s[OF gtrp2gtr-s[OF A]] by (unfold \alpha_nl-def)

8.5.3 Interleaving theorem

In this section, we show that the consistent interleaving operator describes the intuition behind interleavability of normalized paths. We show: Two paths are simultaneously executable if and only if they are consistently interleaveble and the monitors of the initial configurations are compatible.

The split lemma splits an execution from a context of the form ca + cb into two interleavable executions from ca and cb respectively. While further down we prove this lemma for loc/env-path, which is more general but also more complicated, we start with the proof for paths of the multiset-semantics for illustrating the idea.

lemma (in flowgraph) ntr-split:
!!ca cb. \{(ca+cb,w,c')\}\in trcl (ntr fg); valid fg (ca+cb) \implies
\exists ca' cb' wa wb. c'=ca'+cb' \land
w\in (wa \otimes _{\alpha_n} fg wb) \land
mon-c fg ca \cap (mon-c fg cb \cup mon-wq fg wb) = {} \land
mon-c fg cb \cap (mon-c fg ca \cup mon-wq fg wa) = {} \land
(ca,wa,ca')\in trcl (ntr fg) \land (cb,wb,cb')\in trcl (ntr fg)
proof (induct w) — The proof is done by induction on the path
— If the path is empty, the lemma is trivial
  case Nil thus thesis by (rule exI[of - ca], rule exI[of - cb], intro exI[of - []], auto simp add: valid-unconc)
next
  case (Cons e w) note IHP=this
  — We split a non-empty paths after the first (macro) step
then obtain $ch$ where $SPLIT$: $(ca+ch,e,ch) \in ntr fg$ $(ch,w,c') \in trcl (ntr fg)$ by 
(fast dest: trcl-uncons)

— Pick the stack that made the first step

from $gtrE[OF \ SPLIT(1)]$ obtain $s ce sh ceh$ where $NTRS$: $ca+cb=\{#s\}+ce$ $ch=\{#sh\}+ceh \ ((s,ce),e,(sh,ceh)) \in ntr fg$ .

— And separate the threads that spawned during the first step from the ones that where already there

then obtain $csp$ where $CEHFMT$: $ceh=csp+ce$ $mon-c fg csp=\{\}$ by (auto elim!: ntrs-c-cases-s intro!: c-of-initial-no-mon)

— Needed later: The first macrostep uses no monitors already owned by threads that where already there

from $ntrs-mon-c-no-ctx[OF \ NTRS(3)]$ have $MONED$: $mon-w fg e \cap mon-c fg$ $ce=\{\}$ by (auto simp add: mon-c-unconc)

— Needed later: The intermediate configuration is valid

from $ntr-valid-preserve-s[OF \ SPLIT(1)] \ IHP(3)$ have $CHVALID$: valid $fg ch$ .

— We make a case distinction whether the thread that made the first step was in the left or right part of the initial configuration

from $NTRS(1)[symmetric]$ show $?case$ proof (cases rule: mset-unplusm-dist-cases)

— The first step was on a thread in the left part of the initial configuration

case $left$ note $CASE=this$

— We can write the intermediate configuration so that it is suited for the induction hypothesis

with $CEHFMT$ $NTRS$ have $CHFMT$: $ch=\{(\#sh\})+csp+(ca-\{\#s\})\}+cb$

by (simp add: union-ac)

— and by the induction hypothesis, we split the path from the intermediate configuration

with $IHP(1)$ $SPLIT(2)$ $CHVALID$ obtain $ca' cb' wa wb$ where $IHAPP$:

c'\in ca' + cb' 
$w\in wa \cap an fg wb$

$mon-c fg \ ((\#sh\})+csp+(ca-\{\#s\})\} \cap (mon-c fg cb \cup mon-ww fg wb)=\{\}$

$mon-c fg cb \cap (mon-c fg \ ((\#sh\})+csp+(ca-\{\#s\})\}) \cup mon-ww fg wa)=\{\}$

$\{(\#sh\})+csp+(ca-\{\#s\}), wa, ca') \in trcl (ntr fg)$

$(cb, wb, cb') \in trcl (ntr fg)$

by blast

moreover

— It remains to show that we can execute the first step with the right part of the configuration removed

have $FIRSTSTEP$: $(ca,e,\{\#s\})+csp+(ca-\{\#s\})\} \in ntr fg$

proof —

from $CASE(2)$ have $mon-c fg (ca-\{\#s\}) \subseteq mon-c fg ce$ by (auto simp add: mon-c-unconc)

with $ntrs-xchange-context-s$ $NTRS(3)$ $CEHFMT$ $CASE(2)$ have $(s,ca-\{\#s\}),e, (sh,csp+(ca-\{\#s\}))$

$fg$ by blast

from $gtrI-s[OF \ this]$ $CASE(1)$ show $?thesis$ by (auto simp add: union-assoc)
qed

with IHAPP(5) have \((ca,e\#wa,ca')\) \(\in\) \(\text{trcl}\) \((\text{ntr}\ fg)\) by simp

moreover
— and that we can prepend the first step to the interleaving
have \(e\#w \in e\#ww \otimes_{an} fg\) \(wb\)

proof –
from ntrs-called-mon[\(\text{OF NTRS}(3)\)] have \(\text{fst} (\alpha n\ fg\ e) \subseteq \text{mon-s}\ fg\ sh\).
with IHAPP(3) have \(\text{fst} (\alpha n\ fg\ e) \cap \text{mon-ww}\ fg\) \(w\) = \{\} by (auto simp add: mon-c-unconc)

from cil-an-cons[\(\text{OF IHAPP}(2)\) this] show \(?\text{thesis}\).

qed

moreover
— and that the monitors of the initial context does not interfere
have \(\text{mon-c}\) \(fg\ ca\) \(\cap\) \((\text{mon-c}\) \(fg\ cb\) \(\cup\) \(\text{mon-ww}\) \(fg\) \(wb\)) = \{\} \(\text{mon-c}\) \(fg\) \(cb\) \(\cap\) \((\text{mon-c}\) \(fg\) \(ca\) \(\cup\) \(\text{mon-ww}\) \(fg\) \(e\#wa))\) = \{\}

proof –
from ntr-mon-increasing-s[\(\text{OF FIRSTSTEP}\) IHAPP(3)] show \(\text{mon-c}\) \(fg\ ca\) \(\cap\) \((\text{mon-c}\) \(fg\) \(cb\) \(\cup\) \(\text{mon-ww}\) \(fg\) \(wb\)) = \{\} by auto

from MONED\ CASE\ have \(\text{mon-c}\) \(fg\ cb\) \(\cap\) \(\text{mon-w}\) \(fg\) \(e\) = \{\} by (auto simp add: mon-c-unconc)

with ntr-mon-increasing-s[\(\text{OF FIRSTSTEP}\) IHAPP(4)] show \(\text{mon-c}\) \(fg\ cb\) \(\cap\) \((\text{mon-c}\) \(fg\) \(ca\) \(\cup\) \(\text{mon-ww}\) \(fg\) \(e\#wa))\) = \{\} by auto

qed

ultimately show \(?\text{thesis}\) by blast

next
— The other case, that is if the first step was made on a thread in the right part of the configuration, is shown completely analogously

case right note CASE="this"
with CEHFMT\ NTRS\ have \(\text{CHFMT}: \ ch=ca+({\#sh\})+csp+(cb-{\#s\})\)
by (simp add: union-ac)

with IHAP(1) SPLIT(2) CHVALID obtain \(ca'\ cb '\) \(wa\) \(wb\) where IHAPP:\
\(c'=ca'+cb' \in\wa\otimes_{an} fg\) \(wb\) \(\text{mon-c}\) \(fg\ ca\) \(\cap\) \((\text{mon-c}\) \(fg\) \({\#sh\})+csp+(cb-{\#s\}))\) \(\cup\) \(\text{mon-ww}\) \(fg\) \(wb\)=\{\}

\(\text{mon-c}\) \(fg\) \(({\#sh\})+csp+(cb-{\#s\})) \(\cap\) \((\text{mon-c}\) \(fg\) \(ca\) \(\cup\) \(\text{mon-ww}\) \(fg\) \(wa\))\)=\{\}
\((ca,wa,ca')\) \(\text{trcl}\) \((\text{ntr}\ fg)\) \({\#sh\}+csp+(cb-{\#s\}),wb,cb')\) \(\text{trcl}\) \((\text{ntr}\ fg)\)

by blast

moreover
have FIRSTSTEP: \((cb,e,\{\#sh\}+csp+(cb-{\#s\}))\) \(\in\) \(\text{ntr}\ fg\) proof –

from CASE(2) have \(\text{mon-c}\) \(fg\) \(cb-{\#s\})\) \(\subseteq\) \(\text{mon-c}\) \(fg\ ce\) by (auto simp add: mon-c-unconc)

with ntrs-xchange-context-s NTRS(3) CEHFMT\ CASE(2) have \((s,cb-{\#s\}),e,sh,csp+(cb-{\#s\}))\)

\(fg\) by blast

from gtr1-s[\(\text{OF this}\) CASE(1) show \(?\text{thesis}\) by (auto simp add: union-assoc)

qed

with IHAPP(6) have PA: \((cb,e\#wb,cb')\) \(\in\) \(\text{trcl}\) \((\text{ntr}\ fg)\) by simp

moreover
have \(e\#w \in wa \otimes_{an} fg\) \(e\#wb\)

proof –
from ntrs-called-mon[\(\text{OF NTRS}(3)\)] have \(\text{fst} (\alpha n\ fg\ e) \subseteq\) \(\text{mon-s}\) \(fg\) \(sh\).
with IHAPP(4) have \( \text{fst (an fg e) } \cap \text{mon-ww fg wa } = \{ \} \) by (auto simp add: mon-c-unconc)

from cil-an-cons2[OF IHAPP(2) this] show \( \text{thesis} \).

qed

moreover

have mon-c fg cb \( \cap (\text{mon-c fg ca } \cup \text{mon-ww fg wa}) = \{ \} \) mon-c fg ca \( \cap (\text{mon-c fg cb } \cup \text{mon-ww fg (e#wb)) = \{ \} \)

proof –

from ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(4) show mon-c fg cb \( \cap (\text{mon-c fg ca } \cup \text{mon-ww fg wa}) = \{ \} \) by auto

from MONED CASE have mon-c fg ca \( \cap \) mon-ww fg e = \( \{ \} \) by (auto simp add: mon-c-unconc)

with ntr-mon-increasing-s[OF FIRSTSTEP] IHAPP(3) show mon-c fg ca \( \cap (\text{mon-c fg cb } \cup \text{mon-ww fg (e#wb)) = \{ \} \) by auto

qed

ultimately show \( \text{thesis by blast} \)

qed

qed

The next lemma is a more general version of flowgraph.ntr-split for the semantics with a distinguished local thread. The proof follows exactly the same ideas, but is more complex.

lemma (in flowgraph) ntrp-split:

\[ \exists c1 c2 s' c'. \]

\[ \exists w1 w2 c1' c2'. \]

\[ w \in w1 \otimes \text{map ENV w2} \land \]

\[ c' = c1' + c2' \land \]

\[ ((s,c1),w1,(s',c1')) \in \text{trcl (ntrp fg)} \land \]

\[ ((c2,w2,c2')) \in \text{trcl (ntrp fg)} \land \]

\[ \text{mon-ww fg (map le-rem-s w1) } \cap \text{mon-c fg c2} = \{ \} \land \]

\[ \text{mon-ww fg w2 } \cap \text{mon-c fg ((#s#)+c1) = \{ \} } \]

proof (induct w)

case Nil thus \( \text{thesis by (auto intro: exI[of - []] exI[of - {}])} \)

next

case (Cons ee w) then obtain sh ch where SPLIT: \((s,c1+c2),ee,(sh,ch))\in ntrp fg ((sh,ch),w,(s',c'))\in trcl (ntrp fg) by (fast dest: trcl-weak)

from SPLIT(1) show \( \text{case proof (cases rule: gtrp.cases)} \)

case gtrp-loc then obtain e where CASE: \( ee = \text{LOC e ((s,c1+c2),e,(sh,ch))}\in ntrps fg \) by auto

from ntrps-cases-s[OF CASE(2)] obtain csp where CHFMT: \( ch = (csp+c1)+c2 \land s, s' : csp \implies \exists p u v. s = [\text{entry pf}] \land (u, \text{Spawn p, e}) \in \text{edges fg} \land \text{initialproc fg p by (simp add: union-assoc, blast)} \)

with c-of-initial-no-mon have CSPNOMON: mon-c fg csp = \( \{ \} \) by auto

from ntr-valid-preserve-s[OF gtrl-s, OF CASE(2)] Cons.prems(2) CHFMT have VALID: valid fg ((#sh#)+(csp+c1)+c2) by (simp add: union-ac)

from Cons.hyps[OF VALID, OF CASE(2)] obtain w1 w2 c1' c2' where IHAPP: \( w \in w1 \otimes \text{and fg (map ENV w2) c'} = c1' + c2' ((sh, csp + c1), w1, s', c1')) \in trcl (ntrp fg) \)
(c2, w2, c2') ∈ trel (ntrp fg) mon-ww fg (map le-rem-s w1) ∩ mon-c fg c2 = 
{ } mon-ww fg w2 ∩ mon-c fg {{#sh#} + (csp + c1)} = {} by blast
have ee# ∈ ee#w1 ⊗ env fg (map ENV w2) proof (rule cil-cons1)
from ntrp-mon-env-w-no-ctx[OF SPLIT(2), unfolded mon-env-def] have
mon-ww fg (map le-rem-s (env w)) ∩ mon-s fg sh = {} .
moreover have mon-ww fg w2 ≤ mon-ww fg (map le-rem-s (env w)) proof
from cil-subset-il IHAPP(1) ileq-interleave have map ENV w2 ≤ w by blast
from le-list-filter[OF this] have env (map ENV w2) ≤ env w by (unfold
env-def) blast
hence map ENV w2 ≤ env w by (unfold env-def) simp
from le-list-map[OF this, of le-rem-s] have w2 ≤ map le-rem-s (env w) by simp
thus ?thesis by (rule mon-ww-ileq)
qed
ultimately have mon-ww fg w2 ∩ mon-s fg sh = {} by blast
with ntrp-mon-s[OF CASE2)] CASE(1) show fst (env fg ee) ∩ mon-pl (map (env fg) (map ENV w2)) = {} by (auto simp add: cil-cons-helper)
qed (rule IHAPP(1))
moreover have ((s,c1),ee#w1,(s,c1'))∈trel (ntrp fg) proof –
from ntrp-exchange-context-s[of s c1+c2 e sh csp fg c1] CASE(2) CHFMT(1)
have ((s, c1), e, sh, csp + c1) ∈ ntrp fg by (auto simp add: mon-c-unconc
union-ac)
with CASE(1) have ((s, c1), ee, sh, csp + c1) ∈ ntrp fg by (auto intro:
ggrp,ggrp-loc)
also note IHAPP(3)
finally show ?thesis .
qed
moreover from CASE(1) ntrp-mon-e-no-ctx[OF CASE2)] IHAPP(5) have
mon-ww fg (map le-rem-s (ee#w1)) ∩ mon-c fg c2 = {} by (auto simp add:
mon-c-unconc)
moreover from ntrp-mon-increasing-s[OF CASE2)] CHFMT(1) IHAPP(6)
have mon-ww fg w2 ∩ mon-c fg {{#sh#} + c1} = {} by (auto simp add: mon-c-unconc
moreover note IHAPP(2,4)
ultimately show ?thesis by blast
next
case ggrp-env then obtain e ss ce ssh ceh where CASE: ee=ENV e c1+c2={{#ss#}+ce
sh=ss ch={{#ssh#}+ceh ((ss,{{#sh#}+ceh),e,{{ssh,{{#sh#}+ceh})∈ntrp fg by autorom ntrp-cases-s[OF CASE3)] obtain csp where HFMT: {{#sh#}+ceh =
csp + (({{#s#}+ceh \& s :# csp ⇒ ∃ p u v. s = [entry #fg p] (u, Spawn p, v)
in edges fg \& initialproc fg p by (blast)
from union-left-cancel[of {#{#s#} ceh csp+ceh] HFMT(1) have CEHFMT:
kehr=csp+ceh by (auto simp add: union-ac)
from HFMT(2) have CHMON: mon-c fg csp = {} by (blast intro:
c-of-initial-no-mon)
from CASE[2] symmetric show ?thesis proof (cases rule: mset-unplus-dist-cases)
— Made an env-step in c1, this is considered the „left” part. Apply induction
hypothesis with original(!) local thread and the spawned threads on the left side

case left
  with HFMT(1) CASE(4) CEHFMT have CHFMT’: \( ch=(\text{csp} +\{\#\text{ssh}\#\} + (c1 - \{\#\text{ss}\#\})) \) + c2 by (simp add: union-ac)
  have VALID: valid fg \( (\{\#\text{ssh}\#\} + \{\text{csp} + \{\#\text{ssh}\#\} + (c1 - \{\#\text{ss}\#\})) + c2) \)
proof –
  from ntr-valid-preserves-\{\text{OF gtrI-s, OF CASE(5)}\} Cons.prems(2) CASE(2)
  have valid fg \( (\{\#\text{ssh}\#\} + (\{\#\text{s}\#\} + ceh)) \) by (simp add: union-assoc) (auto simp add: union-ac)
  with left CEHFMT show \( \text{?thesis} \) by (auto simp add: union-ac)
qed

from Cons.hyps(\{\text{OF - VALID, of } s' \text{ c}'\} CHFMT' SPLIT(2) CASE(3)) obtain
  \( (s, \text{csp} + \{\#\text{ssh}\#\} + (c1 - \{\#\text{ss}\#\})) \) ∈ ntrp fg proof –
  from \text{left HFMT(1)} have \( \{\#\text{ssh}\#\} + \text{ce} = (\{\#\text{ssh}\#\} + (c1 - \{\#\text{ss}\#\})) + c2 + \{\#\text{s}\#\} + ceh \)
  = \( \text{csp} + \{\#\text{ssh}\#\} + (c1 - \{\#\text{ss}\#\}) + c2 + \{\#\text{s}\#\} + ceh \) by (simp-all add: union-ac)
  with \text{CASE(5) ntrp-exchange-context-s[OF ss \{\#\text{ssh}\#\} + (c1 - \{\#\text{ss}\#\}) + c2 e ssh csp fg (\{\#\text{s}\#\} + (c1 - \{\#\text{ss}\#\}))]) have
  \( (ss, \{\#\text{s}\#\} + (c1 - \{\#\text{ss}\#\}), \text{ssh}, \{\#\text{ssh}\#\} + (\text{csp} + (c1 - \{\#\text{ss}\#\}))) \) ∈ ntrp fg by (auto simp add: mon-c-uncoc union-ac)
  from gtrp.gtrp-\{\text{OF this}\} left(1)[symmetric] CASE(1) show \( \text{?thesis} \) by
  (simp add: union-ac)
qed

from trcl.cons(\{\text{OF this IHAPP(3)}\} have \( (s, c1), ee \text{ #} w1, s', c1' \) ∈ trcl (ntrp fg) .
moreover
  from \text{left} \{\text{ntrp-mon-c-no-ctx[OF CASE(5)] left CASE(1) IHAPP(5)}\} have \text{mon-ww fg (map le-rem-s (ee#w1)) ∩ mon-c fg c2} = \{\} by (auto simp add: mon-c-uncoc)
  moreover
  from \text{ntrp-mon-increasing-s[OF SS]} IHAPP(6) have \text{mon-ww fg w2 ∩ mon-c fg (\{\#\text{s}\#\} + c1) = \{\}} by (auto simp add: mon-c-uncoc)
  moreover note IHAPP(2,4)
ultimately show \( \text{?thesis} \) by blast
next
— Made an env-step in c2. This is considered the right part. Induction hypothesis is applied with original local thread and the spawned threads on the right side

case right
  with HFMT(1) CASE(4) CEHFMT have CHFMT': ch = c1 + (csp + {#ssh#} + (c2 - {#ss#}))
  by (simp add: union-ac)
  have VALID: valid fg ({#s#} + c1 + ((csp + {#ssh#} + (c2 - {#ss#}))))
proof –
  from ntr-valid-preserve-s[OF gtl-s, OF CASE(5)] Cons.prems(2) CASE(2)
  have valid fg ({#ssh#} + ({#s#} + ceh)) by (simp add: union-assoc) (auto simp add: union-ac)
  with right CEHFMT show ?thesis by (auto simp add: union-ac)
qed

from Cons.hyps[OF - VALID, OF s' c'] CHFMT' SPLIT(2) CASE(3) obtain w1 w2 c1' c2' where IHAPP: w ∈ w1 ⊗mon fg map ENV w2 c' = c1' + c2'
  ((s, c1), w1, s', c1') ∈ trcl (ntrp fg) (csp + {#ssh#} + (c2 - {#ss#})),
  w2, c2' ∈ trcl (ntr fg)
  valid-wg (map le-rem-s w1) ∩ mon-c fg (csp + {#ssh#} + (c2 - {#ss#})) = {}\)
  mon-wg fg w2 ∩ mon-c fg ({#s#} + c1) = {} by blast
  have e # w ∈ w1 ⊗mon fg map ENV (c#w2) proof (simp add: CASE(1), rule cil-cons2)
  from IHAPP(5) have mon-wg fg (map le-rem-s w1) ∩ mon-s fg ssh = {}
  by (auto simp add: mon-c-unconc)
  moreover from ntrs-mon-s[OF CASE(5)] CASE(1) have fst (anl fg ee) ⊆ mon-s fg ssh by auto
  ultimately have fst (anl fg ee) ∩ mon-wg fg (map le-rem-s w1) = {} by auto
  moreover have mon-pl (map (anl fg) w1) = mon-wg fg (map le-rem-s w1) by (unfold anl-def') (simp add: cil-an-cons-helper[symmetric])
  ultimately show fst (anl fg (ENV e)) ∩ mon-pl (map (anl fg) w1) = {} using CASE(1) by auto
  qed (rule IHAPP(1))
  moreover
  have SS: (c2,c,csp + {#ssh#} + (c2 - {#ss#})) ∈ ntrp fg proof —
  from right HFMT(1) have {#s#} + cec = {#s#} + c1 + (c2 - {#ss#}) {#s#} + ceh = csp + (c1 + (c2 - {#ss#}))
  by (simp add: union-ac)
  with CASE(5) ntrs-xchange-context-s[of ss {#s#} + c1 + (c2 - {#ss#})] e ssh csp fg c2 - {#ss#} have
  (ss, (c2 - {#ss#}), e, ssh, csp+ (c2 - {#ss#})) ∈ ntrp fg by (auto simp add: mon-c-unconc union-ac)
  from gtl-s[OF this] right(1)[symmetric] show ?thesis by (simp add: union-ac)
  qed
  from trcl.cons[OF this IHAPP(4)] have (c2, e # w2, c2') ∈ trcl (ntr fg).
  moreover
  from ntrs-mon-increasing-s[of SS] IHAPP(5) have mon-wg fg (map le-rem-s w1) ∩ mon-c fg c2 = {} by (auto simp add: mon-c-unconc)
  moreover
from ntrs-mon-e-no-ctx[of case[5]] right IHAPP(6) have mon-ww fg (e#w2) ∩ mon-c fg (f[[#]] + c1) = {} by (auto simp add: mon-c-unconc)
moreover note IHAPP(2,3)
ultimately show thesis by blast
qed
qed
— Just a check that flowgraph.ntrp-split is really a generalization of flowgraph.ntr-split:

lemma (in flowgraph) ntr-split":
assumes A: (ca+cb,w,c')∈trcl (ntr fg)
and VALID: valid fg (ca+cb)
shows ∃ ca' cb' wa wb. c' = ca'+cb' ∧
w ∈ (wa⊗αn fg wb) ∧
mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb) = {}
mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg wa) = {}
(ca,wa,ca')∈trcl (ntr fg) ∧
(cb,wb,cb')∈trcl (ntr fg)
using A VALID by (rule ntr-split)

The unsplit lemma combines two interleavable executions. For illustration purposes, we first prove the less general version for multiset-configurations. The general version for loc/env-configurations is shown later.

lemma (in flowgraph) ntr-unsplit:
assumes A: w∈wa⊗αn fg wb and
B: (ca,wa,ca')∈trcl (ntr fg)
(cb,wb,cb')∈trcl (ntr fg)
mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb)={}
mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg wa)={}
shows (ca+cb,w,ca'+cb')∈trcl (ntr fg)
proof —
— We have to generalize and rewrite the goal, in order to apply Isabelle’s induction method
from A have ∀ ca cb. (ca,wa,ca')∈trcl (ntr fg) ∧ (cb,wb,cb')∈trcl (ntr fg) ∧
mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb)={} ∧ mon-c fg cb ∩ (mon-c fg ca ∪ mon-ww fg wa)={}
— We prove the generalized goal by induction over the structure of consistent interleaving
proof (induct rule: cil-set-induct-fixα)
— If both words are empty, the proposition is trivial
  case empty thus ?case by simp
next —
— The first macrostep of the combined path was taken from the left operand of the interleaving
  case (left e w' w1' w2) thus ?case proof (intro allI impl)
  case (goal1 ca cb) hence I: w' ∈ w1' ⊗αn fg w2 fst (αn fg e) ∩ mon-pl (map
\[(\alpha \, fg) \, w2) = \{\}\]

\[!\alpha cb.
\]

\[
\begin{aligned}
\&[(ca, w1', ca') \in \text{trcl (ntr fg)}; \\
\&(cb, w2, cb') \in \text{trcl (ntr fg)}; \\
\&\text{mon-c fg} \, ca \cap (\text{mon-c fg} \, cb \cup \text{mon-ww fg} \, w2) = \{\}; \\
\&\text{mon-c fg} \, cb \cap (\text{mon-c fg} \, ca \cup \text{mon-ww fg} \, w1') = \{\} \implies \\
\&(ca + cb, w', ca' + cb') \in \text{trcl (ntr fg)} \\
\&(ca, e \neq w1', ca') \in \text{trcl (ntr fg)} \, (cb, w2, cb') \in \text{trcl (ntr fg)} \\
\&\text{mon-c fg} \, ca \cap (\text{mon-c fg} \, cb \cup \text{mon-ww fg} \, w2) = \{\}
\end{aligned}
\]

— Split the left path after the first step

then obtain \(\alpha h\) where \(\text{SPLIT}: (ca,e,cah)\in\text{ntr fg} \, (cah,w1',ca')\in\text{trcl (ntr fg)}\) by (fast dest: \text{trcl-unc}ons)

— and combine the first step of the left path with the initial right context

from \(\text{ntr-add-context-s(OF SPLIT(1)), where } cn=cb\) \(I(7)\) have \((ca + cb, e, \, ca + cb) \in \text{ntr fg} \) by \text{auto}

also

— The rest of the path is combined by using the induction hypothesis

have \((ca + cb, w', ca' + cb') \in \text{trcl (ntr fg)}\) proof —

from \(I(2,6,7) \text{ntr-mon-s(OF SPLIT(1))}\) have MON-CAH: \(\text{mon-c fg} \, cah \cap (\text{mon-c fg} \, cb \cup \text{mon-ww fg} \, w1') = \{\} \) by (cases \(e\)) (auto simp add: \text{cil-\alpha-cons-helper})

with \(I(7)\) have MON-CB: \(\text{mon-c fg} \, cb \cap (\text{mon-c fg} \, cah \cup \text{mon-ww fg} \, w1') = \{\} \) by \text{auto}

from \(I(3)\) (OF SPLIT(2) \(I(5)\) MON-CAH MON-CB) show \(?\text{thesis} \).

qed

next

— The first macrostep of the combined path was taken from the right path — this case is done completely analogous

\[\begin{aligned}
\text{case } (\text{right} \, e \, w' \, w2' \, w1) \text{ thus } ?\text{case proof (intro allI impI)}
\end{aligned}\]

\[\begin{aligned}
\text{case } (\text{goal}1 \, ca \, cb) \text{ hence } I: \, w' \in w1 \otimes_{\alpha \, fg} w2' \, \text{fst (\alpha \, fg} \, e) \cap \text{mon-pl (map (\alpha \, fg) \, w1) = \{}}\]

!!ca cb.

\[
\begin{aligned}
\&[(ca, w1, ca') \in \text{trcl (ntr fg)}; \\
\&(cb, w2', cb') \in \text{trcl (ntr fg)}; \\
\&\text{mon-c fg} \, ca \cap (\text{mon-c fg} \, cb \cup \text{mon-ww fg} \, w2') = \{\}; \\
\&\text{mon-c fg} \, cb \cap (\text{mon-c fg} \, ca \cup \text{mon-ww fg} \, w1) = \{\} \implies \\
\&(ca + cb, w', ca' + cb') \in \text{trcl (ntr fg)} \\
\&(ca, w1, ca') \in \text{trcl (ntr fg)} \, (cb, e\neq w2', cb') \in \text{trcl (ntr fg)} \\
\&\text{mon-c fg} \, ca \cap (\text{mon-c fg} \, cb \cup \text{mon-ww fg} \, (e\neq w2')) = \{\}
\end{aligned}
\]

— the left path after the first step

then obtain \(\beta bh\) where \(\text{SPLIT}: (cb,e,cbh)\in\text{ntr fg} \, (cbh,w2',cb')\in\text{trcl (ntr fg)}\) by (fast dest: \text{trcl-unc}ons)

from \(\text{ntr-add-context-s(OF SPLIT(1)), where } cn=ca\) \(I(6)\) have \((ca + cb, e, \, ca + cbh) \in \text{ntr fg} \) by (auto simp add: \text{union-commute})

also

have \((ca + cbh, w', ca' + cb') \in \text{trcl (ntr fg)}\) proof —
from I\{2,6,7\} ntr-mon-s[OF SPLIT(1)] have MON-CBH: mon-c fg cbh ∩
(mon-c fg ca ∪ mon-ww fg wb) = {} by (cases e) (auto simp add: cil-an-cons-helper)
with I\{6\} have MON-CA: mon-c fg ca ∩ (mon-c fg cbh ∪ mon-ww fg wb')
= {} by auto
from I\{3\}[OF I\{4\} SPLIT(2) MON-CA MON-CBH] show thesis .
qed
finally show case .
qed
qed
with B show thesis by blast

lemma (in flowgraph) ntrp-unsplit:
assumes A: w∈wa⊗and fg (map ENV wb) and
B: ((s,ca),wa,(s',ca'))∈trcl (ntrp fg)
(c,b,w,b',c)∈trcl (ntrp fg)
mon-c fg {{s#} + ca} ∩ (mon-c fg cb ∪ mon-ww fg wb)={} mon-c fg cb ∩ (mon-c fg {{s#} + ca} ∪ mon-ww fg (map le-rem-s wa))={}
shows ((s,ca+cb),w,(s',ca'+cb'))∈trcl (ntrp fg)

proof -
{ fix wb'
  have w∈wa⊗and fg wb' ⟹
    ∀ s ca cb wb. wb'=map ENV wb ∧
    ((s,ca),wa,(s',ca'))∈trcl (ntrp fg) ∧ (c,b,w,b',c)∈trcl (ntrp fg) ∧ mon-c fg
    ({{s#} + ca} ∩ (mon-c fg cb ∪ mon-ww fg wb))={} ∧ mon-c fg cb ∩ (mon-c fg
    ({{s#} + ca} ∪ mon-ww fg (map le-rem-s wa)))={}
    ⟹
    ((s,ca+cb),w,(s',ca'+cb'))∈trcl (ntrp fg)
  proof (induct rule: cil-set-induct-fixa)
    case empty thus case by simp
  next
case (left e w' w1' w2) thus ?case proof (intro allI impI)
    case (goal1 s ca cb wb) hence I: w' ∈ w1' ⊗and fg w2 fst (and fg e) ∩
    mon-pl (map (and fg) w2) = {}!
    s ca cb wb. [!
      w2 = map ENV wb;
      ((s, ca), w1', s', ca') ∈ trcl (ntrp fg);
      (c,b,w,b',c)∈trcl (ntrp fg);
      mon-c fg ({{s#} + ca} ∩ (mon-c fg cb ∪ mon-ww fg wb)) = {};
      mon-c fg cb ∩ (mon-c fg ({{s#} + ca} ∪ mon-ww fg (map le-rem-s
      w1'))) = {}]
  ] ⟹ ((s, ca + cb), w', s', ca' + cb') ∈ trcl (ntrp fg)
  w2 = map ENV wb
  ((s, ca), e # w1', s', ca') ∈ trcl (ntrp fg)
  (c,b,w,b',c)∈trcl (ntrp fg)
  mon-c fg ({{s#} + ca} ∩ (mon-c fg cb ∪ mon-ww fg wb)) = {}
  mon-c fg cb ∩ (mon-c fg ({{s#} + ca} ∪ mon-ww fg (map le-rem-s (e #
  w1')))) = {}
  by blast+}
then obtain \(sh\ cah\) where \(SPLIT\): \(((s, ca), e, (sh, cah)) \in \text{ntrp}\ fg\ ((sh, cah), \text{w1}', (s', ca')) \in \text{trcl} (\text{ntrp}\ fg)\) by \(\text{fast}\ dest: \text{trcl-uncons}\)

from \text{ntrp-add-context-s}[\text{OF}\ SPLIT(1), \text{of}\ cb]\ I(8) have \(((s, ca + cb), e, sh, cah + cb) \in \text{ntrp}\ fg\) by \text{auto}

also have \(((sh, cah + cb), w', (s', ca' + cb')) \in \text{trcl} (\text{ntrp}\ fg)\) proof \(\text{rule}\ I(3)\)

from \text{ntrp-mon-s}[\text{OF}\ SPLIT(1)] I(2,4,7,8) show 1: \(\text{mon-c}\ fg\ (\{\#s\#\} + cah) \cap (\text{mon-c}\ fg\ cb \cup \text{mon-ww}\ fg\ \text{w}b) = \{\}\)

by \(\text{cases}\ e\) \(\text{(rename-tac a, case-tac a, simp add: cil-avn-cons-helper,}

\text{fastforce simp add: cil-avn-cons-helper)+}\)

from I(8) I show \(\text{mon-c}\ fg\ cb \cap (\text{mon-c}\ fg\ (\{\#s\#\} + cah) \cup \text{mon-ww}\ fg\ (\text{map}\ \text{le-rem-s}\ \text{w}1')) = \{\}\) by \text{auto}

\text{qed (auto simp add: I(4,6) SPLIT(2))}

finally show \(?case\).

\text{qed}

next

case \((\text{right}\ ee\ w'\ w2'\ w1)\) thus \(?case\ proof\ \text{(intro allI impI)}\)

case \((\text{goal1}\ s\ ca\ cb\ w)\ )\ hence\ : w' \in w1 \otimes \text{avl}\ fg\ w2'\ \text{fst}\ (\text{avl}\ fg\ ee)\ \cap\ \text{mon-pl}\ (\text{map}\ (\text{avl}\ fg)\ w1) = \{\}\)

!!s\ ca\ cb\ w. [w2' = \text{map}\ \text{ENV}\ w; ((s, ca), w1, s', ca') \in \text{trcl} (\text{ntrp}\ fg); (cb, wb, cb') \in \text{trcl} (\text{ntrp}\ fg); \text{mon-c}\ fg\ (\{\#s\#\} + ca) \cap (\text{mon-c}\ fg\ cb \cup \text{mon-ww}\ fg\ \text{w}b) = \{\}; \text{mon-c}\ fg\ cb \cap (\text{mon-c}\ fg\ (\{\#s\#\} + ca) \cup \text{mon-ww}\ fg\ (\text{map}\ \text{le-rem-s}\ \text{w}1)) = \{\}\]

by \text{fastforce+}

from I(4) obtain e\ wb'\ where\ \text{EE}: w'b = e\#w'b'\ ee = \text{ENV}\ e\ w2' = \text{map}\ \text{ENV}\ \text{w}b'\ by\ \text{(cases}\ wb,\ \text{auto})

with I(6) obtain cbh\ where\ \text{SPLIT}: (cb, e, cbh) \in \text{ntrp}\ fg\ (cbh, wb, cb') \in \text{trcl} (\text{ntrp}\ fg)\) by \(\text{fast dest: trcl-uncons}\)

have \(((s, ca + cb), e, (s, ca + cbh)) \in \text{ntrp}\ fg\) proof -

from \text{gtrE}[\text{OF}\ SPLIT(1)]\ obtain\ \text{sb}\ ceb\ sbh\ cebh\ where\ \text{NTRS}:\ cb = \{\#sb\#\} + ceb\ cbh = \{\#sbh\#\} + cbh\ ((sb, ceb), e, sbh, cebh) \in \text{ntrs}\ fg\ .

from \text{ntrp-add-context-s}[\text{OF}\ \text{NTRS}(3),\ \text{of}\ \{\#s\#\} + ca]\ \text{EE(1)}\ I(7)\ have\ ((sb, \{\#s\#\} + (ca + cebh)), e, sbh, \{\#s\#\} + (ca + cebh)) \in \text{ntrs}\ fg\ by\ \text{(auto simp add: union-ac)}

from \text{gtrp-env}[\text{OF}\ \text{this}]\ \text{NTRS(1,2)}\ EE(2)\ show\ \text{?thesis}\ by\ \text{(simp add: union-ac)}

\text{qed}

also have \(((s, ca + cbh), w', (s', ca' + cb')) \in \text{trcl} (\text{ntrp}\ fg)\) proof \(\text{rule}\ I(3)\)

from \text{ntrp-mon-s}[\text{OF}\ SPLIT(1)] I(2,4,7,8) EE(2) show 1: \(\text{mon-c}\ fg\ cbh \cap (\text{mon-c}\ fg\ (\{\#s\#\} + ca) \cup \text{mon-ww}\ fg\ (\text{map}\ \text{le-rem-s}\ \text{w}1)) = \{\}\)

82
by (cases e) (simp add: cil-αnl-cons-helper, fastforce simp add: cil-αnl-cons-helper)

from I(7) I EE(1) show mon-c fg (\{\#s\#\} + ca) ∩ (mon-c fg cbh ∪ mon-ww fg wb) = {} by auto
qed (auto simp add: EE(3) I(5) SPLIT(2))
finally show ?case.
qed
qed

And finally we get the desired theorem: Two paths are simultaneously executable if and only if they are consistently interleavable and the monitors of the initial configurations are compatible. Note that we have to assume a valid starting configuration.

\textbf{Theorem (in flowgraph)} \texttt{ntr-interleave}: \texttt{valid fg (ca+cb) \implies \exists ca' cb' wa wb.}

\texttt{c' = ca' + cb' ∧ w ∈ (wa ⊗ αn fg wb) ∧ mon-c fg ca ∩ (mon-c fg cb ∪ mon-ww fg wb) = {} ∧ mon-c fg ch ∩ (mon-c fg ca ∪ mon-ww fg wa) = {} ∧ (ca,wa,ca') ∈ trcl (ntr fg) ∧ (cb,wb,cb') ∈ trcl (ntr fg)} \texttt{by (intro iffI)}
\texttt{apply (blast intro: ntrp-split ntrp-unsplit)}

The next is a corollary of \texttt{flowgraph.ntrp-unsplit}, allowing us to convert a path to loc/env semantics by adding a local stack that does not make any steps.

\textbf{Corollary (in flowgraph)} \texttt{ntr2ntrp}: \texttt{[} (c,w,c') ∈ trcl (ntr fg);
\[
\text{mon-c fg \{(\#s\#)+cl\} \cap (\text{mon-c fg c} \cup \text{mon-ww fg w}) = \{\}}
\]
\[
\implies ((s, cl+c), \text{map ENV w}, (s, cl+c')) \in \text{trcl (ntrp fg)}
\]
\text{using ntrp-unsplit[where wa=[], simplified] by fast}

8.5.4 Reverse splitting

This section establishes a theorem that allows us to find the thread in the original configuration that created some distinguished thread in the final configuration.

\text{lemma (in flowgraph) ntr-reverse-split: ![w s' ce']. [}
\text{(c, w, \{\#s'\#\} + ce') \in \text{trcl (ntr fg);}}
\text{valid fg c} \implies
\exists s \text{ w1 w2 ce1'} ce2'.
\text{c=\{\#s\#\} + ce} \land
\text{ce'=ce1'+ce2'} \land
\text{w \in w1} \otimes \alpha \text{on fg w2} \land
\text{mon-s fg s} \cap (\text{mon-c fg ce} \cup \text{mon-ww fg w2}) = \{\} \land
\text{mon-c fg cc} \cap (\text{mon-s fg s} \cup \text{mon-ww fg w1}) = \{\} \land
\text{\{(\#s\#\}, w1, \{\#s'\#\} + ce1') \in \text{trcl (ntr fg)} \land
\text{(ce, w2, ce2')} \in \text{trcl (ntr fg)}
\]

— The proof works by induction on the initial configuration. Note that configurations consist of finitely many threads only
— FIXME: An induction over the size (rather then over the adding of some fixed element) may lead to a smoother proof here

\text{proof (induct c rule: multiset-induct')}
— If the initial configuration is empty, we immediately get a contradiction
\text{case empty hence False by auto thus ?case ..}

next
— The initial configuration has the form \{\#s\#\} + ce.
\text{case (add ce s) }
— We split the path by this initial configuration
\text{from ntr-split[OF add.prems(1,2)] obtain ce1' ce2' w1 w2 where
SPLIT: \{(\#s\#\} + ce'=ce1'+ce2' \in w1} \otimes \alpha \text{on fg w2
mon-c fg cc} \cap (\text{mon-s fg s} \cup \text{mon-ww fg w2}) = \{\} \land
\text{mon-c fg cc} \cap (\text{mon-s fg s} \cup \text{mon-ww fg w1}) = \{\} \land
\text{\{(\#s\#\}, w1, \{\#s'\#\} + ce1') \in \text{trcl (ntr fg)} \land
(ce, w2, ce2') \in \text{trcl (ntr fg)}
by auto
— And then check whether splitting off s was the right choice
\text{from SPLIT(1) show ?case proof (cases rule: mset-unplusm-dist-cases)
— Our choice was correct, s' is generated by some descendant of s“}
\text{case left
with SPLIT show ?thesis by fastforce}

next
— Our choice was not correct, s' is generated by some descendant of ce
\text{case right with SPLIT(6) have C: (ce, w2, \{\#s'\#\} + (ce2' - \{\#s'\#\}) \in \text{trcl (ntr fg)} by auto
— In this case we apply the induction hypothesis to the path from ce
from add.prems(2) have VALID: valid fg ce mon-s fg s ∩ mon-c fg ce = {} 
by (simp-all add: valid-unconc)
from add.hyps[OF C VALID(1)] obtain st cet w21 w22 ce21' ce22' where 
IHAPP: 
    ce=\{#st\#\} + cet 
    ce2' - \{#s'\#\} = ce21' + ce22'
    w2 \in w21 \circ\alpha_n fg w22 
mon-s fg st ∩ (mon-c fg cet ∪ mon-ww fg w22) = {} 
mon-c fg cet ∩ (mon-s fg st ∪ mon-ww fg w21) = {} 
(\{#s\#\}, w21, \{#s'\#\} + ce21') \in \text{trcl} (ntr fg)
(cet, w22, ce22') \in \text{trcl} (ntr fg) by blast

— And finally we add the path from s again. This requires some monitor sorting 
and the associativity of the consistent interleaving operator.
from cil-assoc2 [of w w1 - w2 w22 w21] SPLIT(2) IHAPP(3) obtain w1 where 
CASSOC: w \in w21 \circ\alpha_n fg w1 w21 \circ\alpha_n fg w22 by (auto simp add: cil-commute)
from CASSOC IHAPP(1,3,4,5) SPLIT(3,4) have COMBINE: (\{#s\#\} + cet, w1, ce1' + ce22') \in \text{trcl} (ntr fg) by (rule-tac ntr-unsplit[OF CASSOC(2) 
SPLIT(5) IHAPP(7)]) (auto simp add: mon-c-unconc mon-ww-cil)
moreover from CASSOC IHAPP(1,3,4,5) SPLIT(3,4) have mon-s fg st ∩ 
(mon-c fg (\{#s\#\} + cet) ∪ mon-ww fg w1) = {} mon-c fg (\{#s\#\} + cet) ∩ (mon-s 
fg st ∪ mon-ww fg w21) = {} by (auto simp add: mon-c-unconc mon-ww-cil)
moreover from right IHAPP(1,2) have \{#s\#\} + ce=\{#st\#\} + (\{#s\#\} + cet) 
ce' = ce21' + (\{#s'\#\} + ce22') by (simp-all add: union-ac)
moreover note IHAPP(6) CASSOC(1)
ultimately show ?thesis by blast
qed
qed

end

9 Constraint Systems

theory ConstraintSystems
imports Main AcquisitionHistory Normalization
begin
In this section we develop a constraint-system-based characterization of our analysis.

Constraint systems are widely used in static program analysis. There least 
solution describes the desired analysis information. In its generic form, a 
constraint system \( R \) is a set of inequations over a complete lattice \((L, \sqsubseteq)\) 
and a set of variables \( V \). An inequation has the form \( R[v] \sqsupseteq \text{rhs} \), where 
\( R[v] \in V \) and \( \text{rhs} \) is a monotonic function over the variables. Note that 
for program analysis, there is usually one variable per control point. The 
variables are then named \( R[v] \), where \( v \) is a control point. By standard 
fixed-point theory, those constraint systems have a least solution. Outside

85
the constraint system definition \( R[v] \) usually refers to a component of that least solution.

Usually a constraint system is generated from the program. For example, a constraint generation pattern could be the following:

\[
\text{for } (u, \text{Call } q, v) \in E:
\]

\[
S^k[v] \supseteq \left\{ (\text{mon}(q) \cup M \cup M', \bar{P}) \mid (M, P) \in S^k[u] \land (M', P') \in S^k[r_q] \land \bar{P} \leq P \uplus P' \land |\bar{P}| \leq 2 \right\}
\]

For some parameter \( k \) and a flowgraph with nodes \( N \) and edges \( E \), this generates a constraint system over the variables \( \{S^k[v] \mid v \in N\} \). One constraint is generated for each call edge. While we use a powerset lattice here, we can in general use any complete lattice. However, all the constraint systems needed for our conflict analysis are defined over powerset lattices \((P(\text{type } a), \subseteq)\) for some type \( \text{type } a \). This admits a convenient formalization in Isabelle/HOL using inductively defined sets. We inductively define a relation between variables\(^3\) and the elements of their values in the least solution, i.e. the set \( \{(v, x) \mid x \in R[v]\} \). For example, the constraint generator pattern from above would become the following introduction rule in the inductive definition of the set \( S\text{-cs fg k} \):

\[
\begin{align*}
\text{[(u,Call q,v)}\in\text{edges fg}; (u,M,P)\in S\text{-cs fg k}; \\
\text{return fg q,Ms,Ps)}\in S\text{-cs fg k}; P'&\leq\#P+Ps; \text{size } P' \leq k] \\
\Rightarrow (v,\text{mon fg q } \cup M \cup Ms,P')\in S\text{-cs fg k}
\end{align*}
\]

The main advantage of this approach is that one gets a concise formalization by using Isabelle’s standard machinery, the main disadvantage is that this approach only works for powerset lattices ordered by \( \subseteq \).

9.1 Same-level paths

9.1.1 Definition

We define a constraint system that collects abstract information about same-level paths. In particular, we collect the set of used monitors and all multi-subsets of spawned threads that are not bigger than \( k \) elements, where \( k \) is a parameter that can be freely chosen.

An element \( (u,M,P)\in S\text{-cs fg k} \) means that there is a same-level path from the entry node of the procedure of \( u \) to \( u \), that uses the monitors \( M \) and spawns at least the threads in \( P \).

\[
\text{inductive-set}
\]

\[
S\text{-cs} ::= (n,'p,'ba,'m,'more) \text{ flowgraph-rec-scheme } \Rightarrow \text{ nat } \Rightarrow (n \times 'm \text{ set } \times 'p \text{ multiset } \text{ set})
\]

\(^3\) Variables are identified by control nodes here
for fg k

where

\textit{S-init}: (entry fg p,\{\}\{\#\})\in S-cs fg k

\textit{S-base}: \left\{ (u,\text{Base a},v)\in \text{edges fg}; (u,M,P)\in S-cs fg k \right\} \implies (v,M,P)\in S-cs fg k

\textit{S-call}: \left\{ (u,\text{Call q},v)\in \text{edges fg}; (u,M,P)\in S-cs fg k; (\text{return fg q},M_s,P_s)\in S-cs fg k; P'\leq\#P+P_s; \text{size } P' \leq k \right\} \implies (v,M,P')\in S-cs fg k

\textit{S-spawn}: \left\{ (u,\text{Spawn q},v)\in \text{edges fg}; (u,M,P)\in S-cs fg k; P'\leq\#\{\#q\#\}+P; \text{size } P' \leq k \right\} \implies (v,M,P')\in S-cs fg k

The intuition underlying this constraint system is the following: The \textit{S-init}-constraint describes that the procedures entry node can be reached with the empty path, that has no monitors and spawns no procedures. The \textit{S-base}-constraint describes that executing a base edge does not use monitors or spawn threads, so each path reaching the start node of the base edge also induces a path reaching the end node of the base edge with the same set of monitors and the same set of spawned threads. The \textit{S-call}-constraint models the effect of a procedure call. If there is a path to the start node of a call edge and a same-level path through the procedure, this also induces a path to the end node of the call edge. This path uses the monitors of both path and spawns the threads that are spawned on both paths. Since we only record a limited subset of the spawned threads, we have to choose which of the threads are recorded. The \textit{S-spawn}-constraint models the effect of a spawn edge. A path to the start node of the spawn edge induces a path to the end node that uses the same set of monitors and spawns the threads of the initial path plus the one spawned by the spawn edge. We again have to choose which of these threads are recorded.

9.1.2 Soundness and Precision

Soundness of the constraint system \textit{S-cs} means, that every same-level path has a corresponding entry in the constraint system.

As usual the soundness proof works by induction over the length of execution paths. The base case (empty path) trivially follows from the \textit{S-init} constraint. In the inductive case, we consider the edge that induces the last step of the path; for a return step, this is the corresponding call edge (cf. Lemma \textit{flowgraph.trss-find-call'}). With the induction hypothesis, we get the soundness for the (shorter) prefix of the path, and depending on the last step we can choose a constraint that implies soundness for the whole path.

\textbf{lemma (in flowgraph)} \textit{S-sound}: \forall p v c' P.

\left\{ \left( (\lambda p. \text{entry fg p})\{\#\},w,((v,c'))\right)\in \text{trcl } (\text{trss fg}); \text{size } P\leq k; (\lambda p. \text{entry fg p}) '\# P \leq \# c' \right\} \implies (v,\text{mon-w fg w},P)\in S-cs fg k
proof  (induct w rule: length-compl-rev-induct)
case Nil thus ?case by (auto intro: S-init)
next
case (snoc w e) then obtain sh ch where SPLIT: ([|entry fg p, {#}|], w, (sh, ch)) ∈ trcl (trss fg) ((sh, ch), e, ([v], c′)) ∈ trss fg by (fast dest: trcl-rev-uncons)
from SPLIT(2) show ?case proof (cases rule: trss.cases)
case trss-base then obtain u a where CASE: e = LBase a sh = [u] ch = e′ (u, Base a, v) ∈ edges fg by auto
with snoc.hyps[of w p u e′, OF - - snoc.prems(2, 3)] SPLIT(1) have (u, mon-w fg w, P) ∈ S-cs fg k by blast
moreover from CASE(1) have mon-e fg e = {} by simp
ultimately show ?thesis using S-split[OF CASE(4)] by (auto simp add: mon-w-unconc)
next
case trss-ret then obtain q where CASE: e = LRet sh = return fg q # [v] ch = e′ by auto
with SPLIT(1) have ([|entry fg p, {#}|], w, [return fg q, v], c′) ∈ trcl (trss fg) by simp
from trss-find-call[of this] obtain at ct w1 w2 where FC:
w = w1 @ LCall q # w2
([|entry fg p, {#}|], w1, [|ut, ct|]) ∈ trcl (trss fg)
([|ut, ct|], LCall q ( [|entry fg q, v, ct|]) ∈ trss fg
(at, Call q, v) ∈ edges fg
([|entry fg q, ct|], w2, [|return fg q, c′|]) ∈ trcl (trss fg).
from trss-drop-all-context[of FC(5)] obtain csp' where SLP: e = ct + csp'
([|entry fg q, {#}|], w2, [|return fg q, csp'|]) ∈ trcl (trss fg) by (auto simp add: union-ac)
from FC(1) have LEN: length w1 ≤ length w length w2 ≤ length w by auto
from mset-map-split-orig-le SLP(1) snoc.prems(3) obtain P1 P2 where PSPLIT: P = P1 + P2 (λp. [entry fg p], v) "# P1 ≤ # ct (λp. [entry fg p]) "# P2 ≤ # csp' by blast
with snoc.prems(2) have PSIZE: size P1 ≤ k size P2 ≤ k by auto
from snoc.hyps[of LEN(1) FC(2) SPLIT(1) SPLIT(2)] snoc.hyps[of LEN(2) SLP(2) PSIZE(2) SPLIT(3)] have IHAPP: (ut, mon-w fg w1, P1) ∈ S-cs fg k (return fg q, mon-w fg w2, P2) ∈ S-cs fg k.
from S-call[of FC(4)] IHAPP mset-le-eq-refl[of PSPLIT(1)] snoc.prems(2) FC(1) CASE(1) show (v, mon-w fg (w[@c]), P) ∈ S-cs fg k by (auto simp add: mon-w-unconc Un-ac)
next
case trss-spawn then obtain u q where CASE: e = LSpawn q sh = [u] c′ = [#entry fg q]# + ch (u, Spawn q, v) ∈ edges fg by auto
from mset-map-split-orig-le CASE(3) snoc.prems(3) obtain P1 P2 where PSPLIT: P = P1 + P2 (λp. [entry fg p]) "# P1 ≤ # {#entry fg q}# (λp. [entry fg p]) "# P2 ≤ # ch by blast
with snoc.prems(2) have PSIZE: size P2 ≤ k by simp
from snoc.hyps[of OF - - PSIZE SPLIT(3)] SPLIT(1) CASE(2) have IHAPP: (u, mon-w fg w, P2) ∈ S-cs fg k by blast
have PCOND: P ≤ # {#q#} + P2 proof -
from PSPLIT(2) have P1 ≤ # {#q#} by (auto elim!: mset-le-single-cases mset-map-single-rightE)
with \textit{PSPLIT}(1) show \textit{thesis} by \textit{simp}

\textbf{qed}

from \textit{S-spawn}[OF \textit{CASE}(4) \textit{IHAPP} \textit{PCOND} \textit{snoc.prems}(2)] \textit{CASE}(1) show 
\hspace{0.5cm} (v, \textit{mon-w fg} (w @ \{v\}), P) \in \textit{S-cs fg k} \textit{by (auto simp add: mon-w-unconc)}

\textbf{qed}

Precision means that all entries appearing in the smallest solution of the constraint system are justified by some path in the operational characterization. For proving precision, one usually shows that a family of sets derived as an abstraction from the operational characterization solves all constraints.

In our formalization of constraint systems as inductive sets this amounts to constructing for each constraint a justifying path for the entries described on the conclusion side of the implication – under the assumption that corresponding paths exists for the entries mentioned in the antecedent.

\textbf{lemma (in flowgraph) S-precise:} (v,M,P)\in\textit{S-cs fg k}
\Rightarrow \exists \ p \ c' \ w. \n\hspace{0.5cm} ((\{entry fg p\},\{\#\}),w,\{[v],c'\})\in\textit{trcl} (\textit{trss fg}) \land \n\hspace{0.5cm} size P \leq k \land \n\hspace{0.5cm} (\lambda p. \{entry fg p\}) \ 'P' \leq\# c' \land \n\hspace{0.5cm} M=\textit{mon-w fg} w

\textbf{proof (induct rule: S-cs.induct)}
\hspace{0.5cm} \textbf{case (S-init p)} have 
\hspace{1cm} ((\{entry fg p\},\{\#\}),\{\{}((\{entry fg p\},\{\#\}))\}\in\textit{trcl} (\textit{trss fg}) \textit{by simp-all}
\hspace{1cm} thus \textit{case by fastforce}

\textbf{next}
\hspace{0.5cm} \textbf{case (S-base u a v M P)} then obtain \ p \ c' \ w \ where \textit{IHAPP:} ((\{entry fg p\},\{\#\}),w,\{u\},c') \in \textit{trcl} (\textit{trss fg}) \land \n\hspace{0.5cm} size P \leq k \land \n\hspace{0.5cm} (\lambda p. \{entry fg p\}) \ 'P' \leq\# c' \land \n\hspace{0.5cm} M=\textit{mon-w fg} w \textit{by blast}

\hspace{0.5cm} \textbf{note IHAPP(1)}
\hspace{0.5cm} \textbf{also from} \textit{S-base have} ((\{u\},c'),\textit{LBase a},\{[v],c'\})\in\textit{trss fg} \textit{by (auto intro: trss-base)}
\hspace{0.5cm} \textbf{finally have} 
\hspace{1cm} ((\{entry fg p\},\{\#\}), \textit{w @ [LBase a]}, \{v\}, c') \in \textit{trcl} (\textit{trss fg}) .
\hspace{0.5cm} \textbf{moreover from IHAPP(4) have} M=\textit{mon-w fg} (w @ [LBase a]) \textit{by (simp add: mon-w-unconc)}
\hspace{0.5cm} \textbf{ultimately show} \textit{thesis using IHAPP(2,3,4) by blast}

\textbf{next}
\hspace{0.5cm} \textbf{case (S-call u q v M P Ms Ps P')} then obtain \ p \ csp1 w1 \ where \textit{REACHING-PATH:} ((\{entry fg p\},\{\#\}),w1,\{u\},csp1) \in \textit{trcl} (\textit{trss fg}) \land \n\hspace{0.5cm} size P \leq k \land \n\hspace{0.5cm} (\lambda p. \{entry fg p\}) \ '# P' \leq\# csp1 M = \textit{mon-w fg} w1 \textit{by blast}

\hspace{0.5cm} \textbf{from S-call obtain} \ csp2 w2 \textit{where SL-PATH:} ((\{entry fg q\},\{\#\}),w2,\{\textit{return fg q}\},csp2) \in \textit{trcl} (\textit{trss fg}) \land \n\hspace{0.5cm} size Ps \leq k \land \n\hspace{0.5cm} (\lambda p. \{entry fg p\}) \ '# Ps' \leq\# csp2 Ms = \textit{mon-w fg} w2
\hspace{0.5cm} \textit{by (blast dest: trss-er-path-proc-const)}

\hspace{0.5cm} \textbf{from trss-c-no-mon[OF REACHING-PATH(1)] trss-c-no-mon[OF SL-PATH(1)]}
\hspace{0.5cm} \textbf{have NOMON:} mon-c fg csp1 = \{\} mon-c fg csp2 = \{\} \textit{by auto}
\hspace{0.5cm} \textbf{have} \ ((\{entry fg p\},\{\#\}), \textit{w1 @ LCall q w2 @ LRet},\{[v],csp1+csp2\})\in\textit{trcl} (\textit{trss fg}) \textit{proof}
\hspace{0.5cm} \hspace{0.5cm} \textbf{note REACHING-PATH(1)}
also from trss-call[OF S-call(1)] NOMON have (([u],csp1),LCall q,([entry fg q,v],csp1))∈trss fg by (auto)
  also from trss-add-context[OF trss-stack-comp[OF SL-PATH(1)]] NOMON have (([entry fg q,v],csp1),w2,([return fg q,v],csp1+csp2))∈trcl (trss fg) by (simp add: union-ac)
  also have (([return fg q,v],csp1+csp2),LRet,([v],csp1+csp2))∈trss fg by (rule trss-ret)
  finally show ?thesis by simp
qed

moreover from REACHING-PATH(4) SL-PATH(4) have mon fg q ∪ M ∪ Ms = mon-w fg (w1@LCall q#w2@[LRet]) by (auto simp add: mon-w-unconc)
moreover have (λp. [entry fg p]) ‘# (P)’ ≤# csp1+csp2 (is ?f ‘# P’ ≤# -)
proof –
  from mset-map-le[OF S-call(6)] have ?f ‘# P’ ≤# ?f ‘# P + ?f ‘# Ps by (auto simp add: mset-map-union)
  also from mset-le-mono-add[OF REACHING-PATH(3) SL-PATH(3)] have ... ≤# csp1+csp2 .
  finally show ?thesis .
qed
moreover note S-call(7)
ultimately show ?case by blast
next
  case (S-spawn u q v M P P') then obtain p c' w where IHAPP: (([entry fg p], {#}), w, [u], c') ∈ trcl (trss fg) size P ≤ k (λp. [entry fg p]) ‘# P ≤# c’ M = mon-w fg w by blast
  note IHAPP(1)
  also from S-spawn(1) have (([u],c'),LSpawn q,([v],{#|entry fg q|#}+c'))∈trss fg by (rule trss-spawn)
  finally have (((entry fg p), {#}), w @ [LSpawn q], [v], {#|entry fg q|#} + c') ∈ trcl (trss fg) .
  moreover from IHAPP(4) have M=mon-w fg (w @ [LSpawn q]) by (simp add: mon-w-unconc)
  moreover have (λp. [entry fg p]) ‘# P’ ≤# {#|entry fg q|#} + c’ (is ?f ‘# - ≤# -)
proof –
  from mset-map-le[OF S-spawn(4)] have ?f ‘# P’ ≤# {#|entry fg q|#} + ?f ‘# P by (auto simp add: mset-map-union)
  also from mset-le-mono-add[OF - IHAPP(3)] have ... ≤# {#|entry fg q|#} + c’ by (auto intro: IHAPP(3))
  finally show ?thesis .
qed
moreover note S-spawn(5)
ultimately show ?case by blast
qed

— Finally we can state the soundness and precision as a single theorem

**Theorem (in flowgraph)** S-sound-precise:

$\forall v,M,P \in S-cs \text{ fg k} \iff$

$(\exists p \ c' \ w. ([\text{entry fg p}], \{\#\}, w, (v, c')) \in \text{trcl (trss fg)} \land$ size $P \leq k \land (\lambda p. [\text{entry fg p}]) \ ' # P \leq # c' \land M = \text{mon-w fg w}$
Next, we present specialized soundness and precision lemmas, that reason over a macrostep \((\text{ntrp } fg)\) rather than a same-level path \((\text{trcl } (\text{trss } fg))\). They are tailored for the use in the soundness and precision proofs of the other constraint systems.

**lemma (in flowgraph)** \(S\text{-sound-ntrp}:

**assumes** \(A: (\{[u],[\#]\}, ee,(sh,ch))\in \text{ntrp } fg\) and

**CASE:** \(\forall p\ u' v w.\ [\]

\[\text{eel=LOC } (L\text{Call } p\#u);\]

\((u,\text{Call } p,u')\in \text{edges } fg;\]

\(sh=[v,u'];\]

\(\text{proc-of } fg\ v\ =\ p;\)

\(\text{mon-c } fg\ ch\ =\ \{\};\)

\(!!s.\ s:\# ch \implies \exists p\ u\ v.\ s=[\text{entry } fg\ p]\ \wedge\]

\((u,\text{Spawn } p,v)\in \text{edges } fg\ \wedge\]

\(\text{initialproc } fg\ p;\)

\(!!P.\ (\lambda p.\ [\text{entry } fg\ p])\ \# P \leq \# ch \implies\]

\((v,\text{mon-w } fg\ w,P)\in \text{S-cs } fg\) (size \(P\))

\[\] \implies \)

\(Q\)

\(\text{show}\ Q\)

\(\text{proof} –\)

\(\text{from } A\ \text{obtain } ee\ \text{where } EE: ee=\text{LOC } ee\ ((\{[u],[\#]\}, ee,(sh,ch))\in \text{ntrp } fg\) by\)

\((\text{auto elim: gtrp.cases})\)

\(\text{have } \text{CHFMT} : \forall s.\ s:\# ch \implies \exists p\ u\ v.\ s=[\text{entry } fg\ p]\ \wedge\ (u,\text{Spawn } p,v)\in \text{edges } fg\ \wedge\]

\(\text{initialproc } fg\ p\) by\)

\((\text{auto intro: ntrs.cases.s}[\text{OF } EE(2)])\)

\(\text{with } c\text{-of-initial-no-mon have } \text{CHNOMON: } \text{mon-c } fg\ ch\ =\ \{\}\) by \textbf{blast}

\(\text{from } EE(2)\ \text{obtain } p\ u' v w\ \text{where } \text{FIRSTSPLIT: } ee=L\text{Call } p\#w\ ((\{[u],[\#]\}, L\text{Call } p,([\text{entry } fg\ p],[\#]))\in \text{trss } fg\ sh=[v,u']\ ((([\text{entry } fg\ p],[\#]), w, ([v],ch))\in trcl (\text{trss } fg))\) by\)

\((\text{auto elim: trss.cases})\)

\(\text{from } \text{FIRSTSPLIT} \text{ have } \text{EDGE: } (u,\text{Call } p,u')\in \text{edges } fg\) by\)

\((\text{auto elim!: trss.cases})\)

\(\text{from } \text{trss-bot-proc-const}[\text{where } s=\{\} \text{ and } s'=\{\}, \text{ simplified, } \text{OF } \text{FIRSTSPLIT}(4)]\)

\(\text{have } \text{PROC-OF-V: } \text{proc-of } fg\ v\ =\ p\) by \textbf{simp}

\(\text{have } \forall P.\ (\lambda p.\ [\text{entry } fg\ p])\ \# P \leq \# ch \implies\ (v,\text{mon-w } fg\ w,P)\in \text{S-cs } fg\) (size \(P\))

\(\text{proof} –\)

\(\text{fix } P\ \text{assume } (\lambda p.\ [\text{entry } fg\ p])\ \# P \leq \# ch\)

\(\text{from } \text{S-sound}[\text{OF } \text{FIRSTSPLIT}(4) \text{- this, of size } P]\ \text{show } \?\text{thesis } P\) by \textbf{simp}

\(\text{qed}\)

\(\text{with } EE(1)\ \text{FIRSTSPLIT}(1,3)\ \text{EDGE } \text{PROC-OF-V } \text{CHNOMON } \text{CHFMT} \text{ show}\)

\(Q\) by \((\text{rule-tac CASE})\) \textbf{auto}

\(\text{qed}\)

**lemma (in flowgraph)** \(S\text{-precise-ntrp}:

**assumes** \(\text{ENTRY: } (v,M,P)\in \text{S-cs } fg\ k\) and

\(P: \text{proc-of } fg\ v\ =\ p\) and

\(\text{EDGE: } (u,\text{Call } p,u')\in \text{edges } fg\)

**shows** \(\exists w\ ch.\)

\((([u],[\#]), \text{LOC } (L\text{Call } p\#w),([v,u'],ch))\in \text{ntrp } fg\) (size \(P\) \(\leq k\))

\(91\)
\[ M = \text{mon-w fg w} \land \]
\[ \text{mon-n fg v} = \text{mon fg p} \land \]
\[ (\lambda p. [\text{entry fg p}])' \# P \leq \# ch \land \]
\[ \text{mon-c fg ch} = \{\} \]

proof –

from \( P \) \text{S-precise}[\text{OF ENTRY, simplified}] \text{trss-bot-\text{proc-const}[where s=\[]} \text{and s'=\[]}, \text{simplified}] \text{obtain} \text{usl ch where}

\text{SLPATH: } (([\text{entry fg p}], \{\#\}), \text{usl}, [v], ch) \in \text{trcl (trss fg) size P} \leq k (\lambda p. [\text{entry fg p}])' \# P \leq \# ch M = \text{mon-w fg usl by fastforce}

from \text{mon-n-same-proc}[\text{OF trss-bot-\text{proc-const}[where s=\[]} \text{and s'=\[]}, \text{simplified}, \text{OF SLPATH(1)}]] \text{have} \text{MON-V: mon-n fg v} = \text{mon fg p} \text{by (simp)}

from \text{trss-c-cases}[\text{OF SLPATH(1), simplified}] \text{have} \text{CHFMT: } \bigwedge s s': \# ch \Rightarrow \exists p. s = [\text{entry fg p}] \land (\exists u v. (u, \text{Spawn p}, v) \in \text{edges fg}) \land \text{initialproc fg p} \text{by blast}

with \text{c-of-initial-no-mon have} \text{CHNOMON: mon-c fg ch} = \{\} \text{by blast}

— From the constraints prerequisites, we can construct the first step

\text{have FS: } (([u], \{\#\}), LCall p\#\text{usl}, ([v,u'], ch)) \in \text{trss fg} \text{proof (rule ntrs-step[where r=\[]}, \text{simplified})}

from \text{EDGE show} (([u], \{\#\}), LCall p, [\text{entry fg p}, u'], \{\#\}) \in \text{trss fg by (auto intro: trss-call)}

\text{qed (rule SLPATH(1))}

\text{hence FSP: } (([u], \{\#\}), \text{LOC (LCall p\#\text{usl}, ([v,u'], ch)) \in ntrp fg by (blast intro: gtrp-loc)}

\text{from FSP SLPATH(2,3,4) CHNOMON MON-V show} \text{thesis by blast}

qed

9.2 Single reaching path

In this section we define a constraint system that collects abstract information of paths reaching a control node at \( U \). The path starts with a single initial thread. The collected information are the monitors used by the steps of the initial thread, the monitors used by steps of other threads and the acquisition history of the path. To distinguish the steps of the initial thread from steps of other threads, we use the loc/env-semantics (cf. Section 5.4).

9.2.1 Constraint system

An element \((u, Ml, Me, h) \in RU-cs fg U\) corresponds to a path from \{\#u\#\} to some configuration at \( U \), that uses monitors from \( Ml \) in the steps of the initial thread, monitors from \( Me \) in the steps of other threads and has acquisition history \( h \).

Here, the correspondence between paths and entries included into the inductively defined set is not perfect but strong enough for our purposes: While each constraint system entry corresponds to a path, not each path corresponds to a constraint system entry. But for each path reaching a configuration at \( U \), we find an entry with less or equal monitors and an acquisition history less or equal to the acquisiton history of the path.
The constraint system works by tracking only a single thread. Initially, there is just one thread, and from this thread we reach a configuration at $U$. After a macrostep, we have the transformed initial thread and some spawned threads. The key idea is, that the actual node $U$ is reached by just one of these threads. The steps of the other threads are useless for reaching $U$. Because of the nice properties of normalized paths, we can simply prune those steps from the path.

The $RU$-init-constraint reflects that we can reach a control node from itself with the empty path. The $RU$-call-constraint describes the case that $U$ is reached from the initial thread, and the $RU$-spawn-constraint describes the case that $U$ is reached from one of the spawned threads. In the two latter cases, we have to check whether prepending the macrostep to the reaching path is allowed or not due to monitor restrictions. In the call case, the procedure of the initial node must not own monitors that are used in the environment steps of the appended reaching path ($\text{mon-n fg u} \cap \text{Me} = \{\}$). As we only test disjointness with the set of monitors used by the environment, reentrant monitors can be handled. In the spawn case, we have to check disjointness with both, the monitors of local and environment steps of the reaching path from the spawned thread, because from the perspective of the initial thread, all these steps are environment steps ($\text{(mon-n fg u} \cup \text{ mon fg p) } \cap (\text{Me} \cup \text{Me}) = \{\}$). Note that in the call case, we do not need to explicitly check that the monitors used by the environment are disjoint from the monitors acquired by the called procedure because this already follows from the existence of a reaching path, as the starting point of this path already holds all these monitors.

However, in the spawn case, we have to check for both the monitors of the start node and of the called procedure to be compatible with the already known reaching path from the entry node of the spawned thread.
9.2.2 Soundness and precision

The following lemma intuitively states: If we can reach a configuration that is at \( U \) from some start configuration, then there is a single thread in the start configuration that can reach a configuration at \( U \) with a subword of the original path.

The proof follows from Lemma \textit{flowgraph.ntr-reverse-split} rather directly.

\begin{verbatim}
lemma (in flowgraph) ntr-reverse-split-atU:
  assumes V: valid fg c and
          A: atU U c' and
          B: (c,w,c')\in trcl (ntr fg)
  shows \( \exists s w' c': s \# c \land w' \preceq w \land c' \preceq \# c' \land
           atU U c' \land (\{\# s\}, w', c')\in trcl (ntr fg)

proof -
  obtain ui r ce' where C'FMT: c' = \{\# ui\# r\#\} + ce' ui\in U by (rule atU-fmt[OF A], simp only: mset-contains-eq) (blast dest: sym)
with ntr-reverse-split[OF - V] B obtain s ce w1 w2 ce1 ce2' where RSLIT: c = \{\# s\} + ce ce' = ce1' + ce2' w = w1\otimes\alpha\cap fg w2 (\{\# s\}, w1, \{\# ui\# r\#\} + ce1') \in trcl (ntr fg) by blast
with C'FMT have s \# c w1 \preceq w \{\# ui\# r\#\} + ce1' \preceq \# c' atU U (\{\# ui\# r\#\} + ce1') by (auto dest: cli-ileq)
with RSLIT(4) show \?thesis by blast
qed
\end{verbatim}

The next lemma shows the soundness of the RU constraint system.

The proof works by induction over the length of the reaching path. For the empty path, the proposition follows by the \textit{RU-init}-constraint. For a non-empty path, we consider the first step. It has transformed the initial thread and may have spawned some other threads. From the resulting configuration, \( U \) is reached. Due to \textit{flowgraph.ntr-split} we get two interleavable paths from the rest of the original path, one from the transformed initial thread and one from the spawned threads. We then distinguish two cases: if the first path reaches \( U \), the proposition follows by the induction hypothesis and the \textit{RU-call} constraint.

Otherwise, we use \textit{flowgraph.ntr-reverse-split-atU} to identify the thread that actually reaches \( U \) among all the spawned threads. Then we apply the induction hypothesis to the path of that thread and prepend the first step using the \textit{RU-spawn}-constraint.

The main complexity of the proof script below results from fiddling with the monitors and converting between the multiset-and loc/env-semantics. Also the arguments to show that the acquisition histories are sound approximations require some space.

\begin{verbatim}
lemma (in flowgraph) RU-sound:
  \( \forall u s' c': [((\{u\},\{\#\}), w, (s', c'))\in trcl (ntrp fg); atU U (\{\# s'\#\} + c')] \)
\end{verbatim}

94
\[ \exists M_l M_e h, (u, M_l, M_e, h) \in RU-CS fg U \land M_l \subseteq \text{mon-loc} fg w \land M_e \subseteq \text{mon-env} fg w \land h \leq \alpha h (\text{map} (\alpha H fg) w) \]

— The proof works by induction over the length of the reaching path

\textbf{proof} (\textit{induct} w rule: \textit{length-compl-induct})

— For a reaching path of length zero, the proposition follows immediately by the constraint \textit{RU-init}

\textbf{case} \textit{Nil} \textbf{thus} \textit{?case} by \textit{auto} (\textit{auto intro: RU-init})

\textbf{next}

\textbf{case} (\textit{Cons} \textit{eel} \textit{wwl})

— For a non-empty path, we regard the first step and the rest of the path

\textbf{then obtain} \textit{sh} \textit{ch} \textbf{where} \textit{SPLIT}:

\[ (((u, \{\#\}), \text{eel}, (sh, ch)) \in \text{ntrp fg} \land ((sh, ch), \text{wwl}, (\text{s}', \text{c}')) \in \text{trcl} (\text{ntrp fg})) \]

by (\textit{fast dest: trcl-uncons})

\textbf{obtain} \textit{p} \textit{u} \textit{v} \textit{w} \textbf{where}

— The first step consists of an initial call and a same-level path

\textit{FS-FMT}: \textit{eel} = \text{LOC} (\text{LCall} p \# \textit{w}) (u, \text{Call} p, u') \in \text{edges} fg \textit{sh} = [v, u']

\textit{proc-of} fg \textit{v} = \textit{p} \textit{mon-c} fg \textit{ch} = \{\}

— The only environment threads after the first step are the threads that were spawned by the first step

\and \textit{CHFMT}: \forall s. s' \# ch \Rightarrow \exists p u v. s' = [\text{entry} fg \ p] \land (u, \text{Spawn} p, v) \in \text{edges} fg \land \text{initialproc} fg \textit{p}

— For the same-level path, we find a corresponding entry in the \textit{S-cs} constraint system

\and \textit{S-ENTRY-PAT}: \forall P. (\lambda p. [\text{entry} fg \ p]) 's \# P \leq \# ch \Rightarrow (v, \text{mon-w} fg w, P) \in \text{S-cs} fg (\text{size} \ P)

by (\textit{rule Ssound-ntr}(\textit{OF SPLIT(1)})) \textit{blast}

\textbf{from} \textit{ntrp-valid-preserve-s}(\textit{OF SPLIT(1)}) \textbf{have} \textit{HVALID}: \textit{valid} fg (\{\#sh\#\} + ch) \textbf{by} \textit{simp}

— We split the remaining path by the local thread and the spawned threads, getting two interleavable paths, one from the local thread and one from the spawned threads

\textbf{from} \textit{ntrp-split}(\textit{where} \textit{?c1.0} = \{\#\}, \textit{simplified}, \textit{OF SPLIT(2)} \textit{ntrp-valid-preserve-s}(\textit{OF SPLIT(1)}), \textit{simplified}) \textbf{obtain} \textit{w1} \textit{w2} \textit{c1'} \textit{c2'} \textbf{where}

\textbf{LESPLIT}:

\[ \text{wwl} \in \text{w1} \otimes \text{and} fg \text{ map ENV} \textit{w2} \]

\[ c' = c1' + c2' \]

\[ ((sh, \{\#\}), w1, s', c1') \in \text{trcl} (\text{ntrp fg}) \]

\[ (ch, w2, c2') \in \text{trcl} (\text{ntrp fg}) \]

\[ \text{mon-ww} fg (\text{map} \ \text{le-rem-s} \ w1) \cap \text{mon-c} fg \ ch = \{\} \]

\[ \text{mon-ww} fg w2 \cap \text{mon-s} fg sh = \{\} \]

by \textit{blast}

— We make a case distinction whether \textit{U} was reached from the local thread or from the spawned threads

\textbf{from} \textit{Cons.prems}(2) \textit{LESPLIT}(2) \textbf{have} \textit{atU U} (((\#s'\#) + c1') + c2') \textbf{by} (\textit{auto simp add: union-ac})
thus \textbf{case proof} (cases rule: atU-union-cases)

\textbf{case left} — \textit{U} was reached from the local thread
from \texttt{cil-ileq[OF LESPLIT(1)]} have \texttt{ILEQ: w1 \leq wwl and LEN: length w1 \leq length wwl by (auto simp add: le-list-length)}
— We can cut off the bottom stack symbol from the reaching path (as always possible for normalized paths)
from \texttt{FS-FMT(3) LESPLIT(3) ntrp-stack-decomp[of v [] [u'] [\#] } w1 s' c1' fg, simplified\textbf{ obtain v' rr where DECOMP: s'=v'\#rr@[u'] }\{([v],[\#]),w1,(v'\#rr,c1')\}\in trcl (ntrp fg) \textbf{by auto}
— This does not affect the configuration being at \textit{U}
from \texttt{atU-exchange-stack left DECOMP(1) have ATU: atU U \{\# v'\#rr\} + c1'}
\textbf{by fastforce}
— Then we can apply the induction hypothesis to get a constraint system entry for the path
from \texttt{Cons.hyps[OF LEN DECOMP(2) ATU]} \textbf{obtain Ml Me h where IHAPP: (v,Ml,Me,h)\in RU-cs fg U Ml \subseteq mon-loc fg w1 Me \subseteq mon-env fg w1 h \leq oah (map (anl fg) w1) by blast}
— Next, we have to apply the constraint \texttt{RU-call}
from \texttt{S-ENTRY-PAT[of \{\#, simplified]} have \texttt{S-ENTRY: (v, mon-w fg w, \{\#\}) \in S-cs fg 0}}
\textbf{have \texttt{MON-U-ME: mon-n fg u \cap Me = \{\} proof —}}
\textbf{from ntrp-mon-env-w-no-ctx[OF Cons.prems(1)] have mon-env fg wwl \cap mon-n fg u = \{\} by (auto)}
\textbf{with \texttt{mon-env-ileq[OF ILEQ] IHAPP(3) show ?thesis by fast}}
\textbf{qed}

\textbf{from RU-call[OF FS-FMT(2,4) S-ENTRY IHAPP(1) MON-U-ME] have (u, mon fg p \cup mon-w fg w \cup Ml, Me, ah-update h (mon fg p, mon-w fg w) (Ml \cup Me)) \in RU-cs fg U}}
— Then we assemble the rest of the proposition, that are the monitor restrictions and the acquisition history restriction
\textbf{moreover have mon-fg p \cup mon-w fg w \cup Ml \subseteq mon-loc fg (eel\#wwl) using mon-loc-ileq[OF ILEQ] IHAPP(2) FS-FMT(1) by fastforce}
\textbf{moreover have Me \subseteq mon-env fg (eel\#wwl) using mon-env-ileq[OF ILEQ, of fg] IHAPP(3) by auto}
\textbf{moreover have ah-update h (mon fg p, mon-w fg w) (Ml \cup Me) \leq oah (map (anl fg) (eel\#wwl)) proof (simp add: ah-update-cons)}
\textbf{show ah-update h (mon fg p, mon-w fg w) (Ml \cup Me) \leq ah-update (oah (map (anl fg) wwl)) (anl fg eel) (mon-pl (map (anl fg) wwl)) proof (rule ah-update-mono)}
\textbf{from IHAPP(4) have h \leq oah (map (anl fg) w1) .}
\textbf{also from oah-ileq[OF le-list-map[OF ILEQ]] have oah (map (anl fg) w1)}
\textbf{\leq oah (map (anl fg) wwl) .}
\textbf{finally show h \leq oah (map (anl fg) wwl) .}
\textbf{next from FS-FMT(1) show (mon fg p, mon-w fg w) = anl fg eel by auto}
\textbf{next from IHAPP(2,3) have (Ml \cup Me) \subseteq mon-pl (map (anl fg) w1) by (auto simp add: mon-pl-of-anl)}
\textbf{also from mon-pl-ileq[OF le-list-map[OF ILEQ]] have \ldots \subseteq mon-pl (map (anl fg) wwl) .}

96
finally show $(M\cup Me) \subseteq mon-pl (map (\alpha n fg) \ wwl)$. 

qed

qed

ultimately show \textit{?thesis} by blast

next

case right — $U$ was reached from the spawned threads

from cil-ileq\{OF LESPLIT\{1\}} le-list-length\{of map ENV w2 wwl\} have ILEQ:\nmap ENV w2\le\ wwl and LEN: \length w2 \le \length wwl by (auto)

from HVALID have CHVALID: valid \fg ch mon-s \fg sh \cap mon-c \fg ch = \{} by (auto simp add: valid-anconc)

— We first identify the actual thread from that $U$ was reached

from ntr-reverse-split-atU\{OF CHVALID\{1\} right LESPLIT\{4\}\} obtain q \wcr \cre' where RI:\{entry \fg q \# \ chi \wle w2 \cre' \le \# c2' atU U \cre' (\#entry \fg q\#),\wcr,\cre'\}\in trcl (ntr fg) by (blast dest: CHFMT)

— In order to apply the induction hypothesis, we have to convert the reaching path to loc/env semantics

from ntrs.gtr2gtrp\{where c=\{\#\}, simplified, OF RI\{5\}\} obtain sr' cre' wwr where RI-\nTRP: \cre'=\{#sr'\#\}+cre' \wrr=map le-rem-s wwr ((\#entry \fg q\},\{\#\}),wwr,(sr',cre')\}\in trcl (ntrp fg) by blast

from LEN le-list-length\{OF RI\{2\}\} RI-\nTRP\{2\} have LEN': \length wwr \le length wwl by simp

— The induction hypothesis yields a constraint system entry

from Cons.hyps\{OF LEN' RI-\nTRP\{3\}\} RI-\nTRP\{1\} RI\{4\} obtain Ml Me h where IHAPP: (entry \fg q, Ml, Me, h)\in RU-cs \fg U Ml \subseteq mon-loc \fg wwr Me \subseteq mon-env \fg wwr h \le cah (map (\alpha n fg) wwr) by auto

— We also have an entry in the same-level path constraint system that contains the thread from that $U$ was reached

from S-ENTRY-PAT\{of \{\#q\#\}, simplified\} RI\{1\} have S-ENTRY: (v, mon-w \fg w, \{\#q\#\}) \in S-cs \fg 1 by auto

— Before we can apply the $RU$-spawn-constraint, we have to analyze the monitors

have MON-MLE-ENV: Ml U Me \subseteq mon-env \fg wwr proof —

from IHAPP\{2,3\} have Ml U Me \subseteq mon-loc \fg wwr U mon-env \fg wwr by auto

also from mon-ww-of-le-rem\{symmetric\} RI-\nTRP\{2\} have \ldots = mon-ww \fg \wlr by fastforce

also from mon-env-ileq\{OF ILEQ\} mon-ww-ileq\{OF RI\{2\}\} have \ldots \subseteq mon-env \fg wwr by fastforce

finally show \textit{?thesis} .

qed

have MON-UP-MLE: (mon-n \fg u \cup mon \fg p) \cap (Ml U Me) = \{} proof —

from ntrp-mon-env-ww-no-ctx\{OF SPLIT\{2\}\} FS-FMT\{3,4\} edges-part\{OF FS-FMT\{2\}\} have (mon-n \fg u \cup mon \fg p) \cap mon-env \fg wwr = \{} by (auto simp add: mon-n-def)

with MON-MLE-ENV show \textit{?thesis} by auto

qed

— Finally we can apply the $RU$-spawn-constraint that yields us an entry for the reaching path from $a$

from RU-spawn\{OF FS-FMT\{2,4\}\} S-ENTRY - IHAPP\{1\} MON-UP-MLE]
have \((u, \text{mon} fg p \cup \text{mon-w} fg w, \text{Me}) \in RU-cs fg U\) by simp

— Next we have to assemble the rest of the proposition

moreover have \(\text{mon} fg p \cup \text{mon-w} fg w \subseteq \text{mon-loc} fg (eel#wwl)\) using FS-FMT(1) by fastforce

moreover have \(\text{Me} \subseteq \text{mon-env} fg (eel#wwl)\) using MON-MLE-ENV by auto

moreover have \(\text{ah-update} h (\text{mon} fg p, \text{mon-w} fg w) (\text{Me}) \leq \alpha h (\text{map} (\text{mon} fg eel) (eel#wwl))\) — Only the proposition about the acquisition histories needs some more work

proof (simp add: ah-update-cons)

have MAP-HELPER: \(\text{map} (\text{mon} fg) wwr \leq \text{map} (\text{mon} fg) wwl\) proof —

from RI-NTRP(2) have \(\text{map} (\text{mon} fg) wwr = \text{map} (\text{mon} fg) wr\) by (simp add: map-in)

also from le-list-map[OF RI(2)] have \(\ldots \leq \text{map} (\text{mon} fg) w2\).

also have \(\ldots = \text{map} (\text{mon} fg) (\text{map} \text{ENV} w2)\) by simp

also from le-list-map[OF ILEQ] have \(\ldots \leq \text{map} (\text{mon} fg) wwl\).

finally show ?thesis.

qed

show \(\text{ah-update} h (\text{mon} fg p, \text{mon-w} fg w) (\text{Me}) \leq \text{ah-update} (\alpha h (\text{map} (\text{mon} fg) wwl)) (\text{mon-pl} (\text{map} (\text{mon} fg) wwl))\) proof (rule ab-update-mono)

from IHAPP(4) have \(h \leq \alpha h (\text{map} (\text{mon} fg) wwr)\).

also have \(\ldots \leq \alpha h (\text{map} (\text{mon} fg) wwl)\) by (rule \alpha h-Ileq[OF MAP-HELPER])

finally show \(h \leq \alpha h (\text{map} (\text{mon} fg) wwl)\).

next

from FS-FMT(1) show \((\text{mon} fg p, \text{mon-w} fg w) = \text{mon} eel\) by simp

next

from IHAPP(2,3) mon-pl-Ileq[OF MAP-HELPER] show \(\text{Me} \subseteq \text{mon-pl} (\text{map} (\text{mon} fg) wwl)\) by (auto simp add: mon-pl-of-mon)

qed

qed

ultimately show ?thesis by blast

qed

Now we prove a statement about the precision of the least solution. As in the precision proof of the S-cs constraint system, we construct a path for the entry on the conclusion side of each constraint, assuming that there already exists paths for the entries mentioned in the antecedent.

We show that each entry in the least solution corresponds exactly to some executable path, and is not just an under-approximation of a path; while for the soundness direction, we could only show that every executable path is under-approximated. The reason for this is that in effect, the constraint system prunes the steps of threads that are not needed to reach the control point. However, each pruned path is executable.

lemma (in flowgraph) RU-precise: \((u, \text{Me}, h) \in RU-cs fg U\) \(
\exists w s' c'. (([u], \#), w,(s', c')) \in \text{trlp} (\text{ntrp} fg) \wedge
\)

98
atU U (\{\#s\#\} + c') ∧
mon-loc fg w = Ml ∧
mon-env fg w = Me ∧
αah (map (and fg) w) = h

proof (induct rule: RU-cs.induct)
— The RU-init constraint is trivially covered by the empty path

case (RU-init u) thus ?case by (auto intro: exI[of _ []])

next
— Call constraint

case (RU-call u p u' v M P M l Me h)
thcn obtain w s' c' where IHAPP: \(((v), \{\#\}), w, s', c' ∈ trcl (ntrp fg) atU U (\{\#s\#\} + c') mon-loc fg w = Ml mon-env fg w = Me αah (map (and fg) w) = h by blast

from RU-call.hyps(2) S-precise[OF RU-call.hyps(3), simplified] trss-bot-proc-cons[where s=[] and s'=[], simplified] obtain wsl ch where
SLPATH: \(((\text{entry fg p}), \{\#\}), \text{wsl}, [v], ch) ∈ trcl (trss fg) M = mon-w fg wsl by fastforce

from trss-cases[OF SLPATH(1), simplified] have CHFMT: \(\exists s. s : \# \ ch \implies \exists p. s = [\text{entry fg p}] \land (3\ u. (u, \text{Spawn p}, v) \in \text{edges fg}) \land \text{initialproc p g by blast}

with c-of-initial-no-mon have CHNOMON: mon-c fg ch = {} by blast
— From the constraints prerequisites, we can construct the first step

have FS: \(((\text{u}), \{\#\}), \text{LCall p\#wsl}, [(v,u'),ch)]\in ntrp fg proof (rule ntrs-step[where r=[]], simplified])

from RU-call.hyps(1) show \(((\text{u}), \{\#\}), \text{LCall p}, [\text{entry fg p}, u'], \{\#\}) \in trss fg by (auto intro: trss-call)

qed (rule SLPATH(1))

hence FSP: \(((\text{u}), \{\#\}), \text{LOC (LCall p\#wsl)}, [(v,u'),ch)]\in ntrp fg by (blast intro: gtrp-loc)

also
— The rest of the path comes from the induction hypothesis, after adding the rest of the threads to the context

have \(((\text{v}), u'), ch), w, s' @ [u'], c' + ch) \in trcl (ntrp fg) proof (rule ntrp-add-context[OF ntrp-stack-comp[OF IHAPP(1), where r=[][u'], where cn=ch, simplified]])

from RU-call.hyps(1,6) IHAPP(4) show mon-n fg u' ∩ mon-env fg w = {} by (auto simp add: mon-n-def edges-part)

from CHNOMON show mon-ww fg (map le-rem-s w) ∩ mon-c fg ch = {} by auto

qed

finally have \(((\text{u}), \{\#\}), \text{LOC (LCall p \# wsl) \# w, s' @ [u'], c' + ch)} \in trcl (ntrp fg).
— It is straightforward to show that the new path satisfies the required properties for its monitors and acquisition history

moreover from IHAPP(2) have atU U (\{\# s'\#[u\#] \#\}+(c'+ch)) by auto

moreover have mon-loc fg \(((\text{LOC (LCall p \# wsl) \# w}) = mon fg p \cup M \cup Ml)

using SLPATH(2) IHAPP(3) by auto

moreover have mon-env fg \(((\text{LOC (LCall p \# wsl) \# w}) = Me using IHAPP(4)

by auto

moreover have αah (map (and fg) (LOC (LCall p \# wsl) \# w)) = ah-update
h (mon fg p, M) (Ml ∪ Me) proof –
  have αah (map (αnl fg) (LOC (LCall p # wsl) # w)) = ah-update (αah (map (αnl fg) w)) (mon fg p, mon-w fg wsl) (mon-pl (map (αnl fg) w)) by (auto simp add: ah-update-cons)
  also have: . . . = ah-update h (mon fg p, M) (Ml ∪ Me) proof –
    from IHAPP(5) have αah (map (αnl fg) w) = h.
  moreover from SLPATH(2) have (mon fg p, mon-w fg wsl) = (mon fg p, M) by (simp add: mon-pl-of-αnl)
  moreover from IHAPP(3,4) have mon-pl (map (αnl fg) w) = Ml ∪ Me by (auto simp add: mon-pl-of-αnl)
  ultimately show ?thesis by simp
  qed
  finally show ?thesis .
  qed
ultimately show ?case by blast

next —
    — Spawn constraint
    case (RU-spawn u p u' v M P q Ml Me h) then obtain w s' c' where IHAPP:
      (((entry fg q), (#)), w, s', c') ∈ trcl (ntrp fg) atU U ((#s'#) + c') mon-loc fg w = Ml mon-env fg w = Me oah (map (αnl fg) w) = h by blast
    from RU-spawn.hyps(2) S-precise[OF RU-spawn.hyps(3), simplified] trss-bot-loc-constr[where s=[] and s'=[], simplified] obtain wsl ch where
      SLPATH: (((entry fg p), (#)), wsl, [v], ch) ∈ trcl (trss fg) M = mon-w fg wsl size P ≤ I (λp. entry fg p) ' # P ≤ # ch by fastforce
      with RU-spawn.hyps(4) obtain ch' where PFMT: P=#q# ch = #entry fg q# ch' = (auto elim!: mset-size-le1-cases mset-le-addE)
      from trss-c-cases[OF SLPATH(1), simplified] have CHNOM: s : # ch' =⇒ ∃p. s = [entry fg p] ∧ (∃u v. (u, Spawn p, v) ∈ edges fg) ∧ initialproc fg p by blast
      with c-of-initial-no-mon have CHNOMON: mon-c fg ch = {} by blast
      have FS: (((u),(#)), LCall p#wsl,([v,u'],ch)) ∈ ntrgs fg proof (rule ntrg-step[where r=[]], simplified])
      from RU-spawn.hyps(1) show (((u), #)), LCall p, [entry fg p, u'], (#) ∈ trss fg by (auto intro: trss-call)
      qed (rule SLPATH(1))
      hence FSP: (((u),(#)), LOC (LCall p#wsl,([v,u'],ch)) ∈ ntrp fg by (blast intro: gtrp-loc)
      also have (((v, u'), ch), map ENV (map le-rem-s w), [v, u'], che+{(#s'#)+c'}) ∈ trcl (ntrp fg) proof –
        from IHAPP(3,4) have mon-w fg (map le-rem-s w) ⊆ Ml ∪ Me by (auto simp add: mon-w-of-le-rem)
        with RU-spawn.hyps(1,2,7) have (mon-n fg v ∪ mon-n fg u') ∩ mon-w fg (map le-rem-s w) = {} by (auto simp add: mon-n-def edges-part)
        with ntr2ntrp[OF gtrp2gtr[OF IHAPP(1)]], of [v,u'] che] PFMT(2) CHNOMON
        show ?thesis by (auto simp add: union-ac mon-c-unc onc)
      qed
      finally have (((u), (#)), LOC (LCall p # wsl) # map ENV (map le-rem-s w), [v, u'], che + {(#s'#) + c'}) ∈ trcl (ntrp fg).
      moreover from IHAPP(2) have atU U ((#|v,u'|#) + (che+{(#s'#) + c'}))
by simp

moreover have mon-loc fg (LOC (LCall p # wsl) # map ENV (map le-rem-s w)) = mon fg p ∪ M using SPATH (2) by (auto simp del: map-map)

moreover have mon-env fg (LOC (LCall p # wsl) # map ENV (map le-rem-s w)) = Ml ∪ Me using IHAPP (3, 4) by (auto simp add: mon-ww-of-le-rem simp del: map-map)

moreover have α ah (map (α nl fg) (LOC (LCall p # wsl) # map ENV (map le-rem-s w))) = ah-update h (mon fg p, M) by (simp add: ah-update-cons o-assoc)

also have . . . = ah-update h (mon fg p, M) (Ml ∪ Me) by (simp add: ah-update-cons o-assoc)

ultimately show ?thesis by simp
qed

ultimately show ?thesis by blast
qed

9.3 Simultaneously reaching path

In this section, we define a constraint system that collects abstract information for paths starting at a single control node and reaching two program points simultaneously, one from a set \( U \) and one from a set \( V \).

9.3.1 Constraint system

An element \((u, Ml, Me) \in RUV-cs fg U V\) means, that there is a path from \{#u#\} to some configuration that is simultaneously at \( U \) and at \( V \). That path uses monitors from \( Ml \) in the first thread and monitors from \( Me \) in the other threads.

inductive-set

\text{RUVCs} :: (\text{n}.'p','ba','m','more') \text{flowgraph-rec-scheme} \Rightarrow
\text{n} set \Rightarrow \text{n} set \Rightarrow (\text{n} \times \text{m set} \times \text{m set}) set

for \( fg U V \)

where

\text{RUVC-all}:
\begin{align*}
& ((u,\text{Call}\ p,\text{w}) \in \text{edges}\ fg;\ \text{proc-of}\ fg\ v = p; (v,\text{M},\text{P}) \in \text{S-cs}\ fg\ 0; \\
& (v,\text{Ml},\text{Me}) \in \text{RUVCs}\ fg\ U\ V;\ \text{mon-n}\ fg\ u \cap \text{Me} = \{\}) \Rightarrow \\
& (u,\text{mon}\ fg\ p \cup \text{M} \cup \text{Ml},\text{Me}) \in \text{RUVCs}\ fg\ U\ V)
\end{align*}

\text{RUVC-spawn}:
\[
\begin{align*}
\{ (u, \text{Call} p, u) \in \text{edges \ fg}; & \; \text{proc\-af \ fg \ v = p}; \; (v, M, P) \in S\text{-cs \ fg} 1; \; q : \# \; P; \\
& (\text{entry} \ fg \ q, Ml, Me) \in RU\text{-cs \ fg} \ U \ V; \\
& (\text{mon-n} \ fg \ u \cup \text{mon} \ fg \ p) \cap (Ml \cup Me) = \{\} \} \\
\implies & \; (u, \text{mon} \ fg \ p \cup M; \; Ml \cup Me) \in RU\text{-cs \ fg} \ U \ V
\end{align*}
\]}

\textbf{RUV-split-le:}
\[
\begin{align*}
\{ (u, \text{Call} p, u) \in \text{edges \ fg}; & \; \text{proc\-af \ fg \ v = p}; \; (v, M, P) \in S\text{-cs \ fg} 1; \; q : \# \; P; \\
& (v, Ml, Me, h) \in RU\text{-cs \ fg} \ U; \; (\text{entry} \ fg \ q, Ml', Me', h') \in RU\text{-cs \ fg} \ V; \\
& (\text{mon-n} \ fg \ u \cup \text{mon} \ fg \ p) \cap (Ml' \cup Me') = \{\}; \; h \ [\ast] \ h' \} \\
\implies & \; (u, \text{mon} \ fg \ p \cup M \cup Ml, Me \cup Ml' \cup Me') \in RU\text{-cs \ fg} \ U \ V
\end{align*}
\]}

\textbf{RUV-split-el:}
\[
\begin{align*}
\{ (u, \text{Call} p, u) \in \text{edges \ fg}; & \; \text{proc\-af \ fg \ v = p}; \; (v, M, P) \in S\text{-cs \ fg} 1; \; q : \# \; P; \\
& (v, Ml, Me, h) \in RU\text{-cs \ fg} \ U; \; (\text{entry} \ fg \ q, Ml', Me', h') \in RU\text{-cs \ fg} \ V; \\
& (\text{mon-n} \ fg \ u \cup \text{mon} \ fg \ p) \cap (Ml \cup Me \cup Ml' \cup Me') = \{\}; \; h \ [\ast] \ h' \} \\
\implies & \; (u, \text{mon} \ fg \ p \cup M \cup Ml, Me \cup Ml' \cup Me') \in RU\text{-cs \ fg} \ U \ V
\end{align*}
\]}

\textbf{RUV-split-ee:}
\[
\begin{align*}
\{ (u, \text{Call} p, u) \in \text{edges \ fg}; & \; \text{proc\-af \ fg \ v = p}; \; (v, M, P) \in S\text{-cs \ fg} 2; \\
& \{\#q\} + \{\#q'\} \leq \# \; P; \\
& (\text{entry} \ fg \ q, Ml, Me, h) \in RU\text{-cs \ fg} \ U; \; (\text{entry} \ fg \ q', Ml', Me', h') \in RU\text{-cs \ fg} \ V; \\
& (\text{mon-n} \ fg \ u \cup \text{mon} \ fg \ p) \cap (Ml \cup Ml' \cup Me \cup Me') = \{\}; \; h \ [\ast] \ h' \} \\
\implies & \; (u, \text{mon} \ fg \ p \cup M, Ml \cup Ml' \cup Me \cup Me') \in RU\text{-cs \ fg} \ U \ V
\end{align*}
\]

The idea underlying this constraint system is similar to the \textit{RU-cs}-constraint system for reaching a single node set. Initially, we just track one thread. After a macrostep, we have a configuration consisting of the transformed initial thread and the spawned threads. From this configuration, we reach two nodes simultaneously, one in \textit{U} and one in \textit{V}. Each of these nodes is reached by just a single thread. The constraint system contains one constraint for each case how these threads are related to the initial and the spawned threads:

\textbf{RUV_call} Both, \textit{U} and \textit{V} are reached from the initial thread.

\textbf{RUV_spawn} Both, \textit{U} and \textit{V} are reached from a single spawned thread.

\textbf{RUV_split_le} \textit{U} is reached from the initial thread, \textit{V} is reached from a spawned thread.

\textbf{RUV_split_el} \textit{V} is reached from the initial thread, \textit{U} is reached from a spawned thread.

\textbf{RUV_split_ee} Both, \textit{U} and \textit{V} are reached from different spawned threads.

In the latter three cases, we have to analyze the interleaving of two paths each reaching a single control node. This is done via the acquisition history information that we collected in the \textit{RU-cs}-constraint system. Note that we do not need an initializing constraint for the empty path, as a single configuration cannot simultaneously be at two control nodes.
9.3.2 Soundness and precision

**Lemma (in flowgraph)** $RUV$-sound: $!!u s' c'$.

\[
\{\text{mon-ww fg w2}}\} \subseteq \text{w1 mon-loc fg w} \land \text{Me} \subseteq \text{mon-env fg w}
\]

— The soundness proof is done by induction over the length of the reaching path

**Proof** (induct \( w \) rule: length-compl-induct)

— In case of the empty path, a contradiction follows because a single-thread configuration cannot simultaneously be at two control nodes

```
case Nil hence False by simp thus \textit{case} ..
```

**Next**

```
case (Cons ee ww) then obtain \( sh \ ch \) where \textit{SPLIT}: \( (((w1,\{\#}\})),ee,(sh,ch))\in ntrp fg ((sh,\textit{ww}),\textit{ww},(s',c'))\in trcl (ntrp fg) \) by (fast dest: trcl-uncons)
```

**From** \( ntrp\{-\textit{split}\} \) \{\textit{fc1.0=\{\#, simplified, OF SPLIT(2) ntrp-valid-preserves-OF SPLIT(1)\}, simplified\} obtain \( w1 w2 c1' c2' \) where \( \textit{LESPLIT}: wu \in w1 \odot a\textit{nf fg}\) map ENV \( w2 c' = c1' + c2'((sh, \{\#\}), w1, s', c1') \in trcl (ntrp fg) (ch, w2, c2') \in trcl (ntrp fg) \) mon-ww \( fg w2 \) (map \( le-rem-s w1 \) \( \cap \) mon-c \( fg ch = \{\} \) mon-ww \( w2 \cap \) mon-s \( fg sh = \{\} \)

by blast

**Obtain** \( p u' v w \) where

**FS-FMT**: \( ee = \textit{LOC} (\textit{LCall p }\# w) (u, \textit{Call p, u'}) \in \textit{edges fg sh} = \{v, u\} \)

**Proc-of fg v = p mon-c fg ch = \{\}**

**and CHFMT**: \( \lambda s, s' : \# ch \Rightarrow \exists p u v. s = [\text{entry fg p}] \land (u, \text{iSpawn p, v}) \in \text{edges fg} \land \text{initialproc fg p} \)

**and S-ENTRY-PAT**: \( \lambda P. (\lambda p. [\text{entry fg p}]) \text{ ' # P }\leq \# ch \Rightarrow (v, \text{mon-w fg w}, P) \in S\{-cs fg\} (\text{size P}) \)

by \{rule S-sound-ntrp[OF SPLIT(1)]\} blast

**From** \( ntrp\{-}\textit{mon-env-w-no-ctx}\{OF SPLIT(2)\} \) \( \text{FS-FMT(3,4)} \) \text{edges-part[OF FS-FMT(2)]} \)

**Have** \( \text{MON-PU}: \text{mon-env fg ww }\cap \) (mon \( fg p \cup \text{mon-n fg w} = \{\} \) \{by (auto simp add: mon-n-def)\}

**From** \( \text{cil-ileq[OF LESPLIT(1)]} \) \text{mon-loc-ileq[of w1 w1 fg]} \( \text{mon-env-ileq[of w1 w1 fg]} \)

**Have** \( \text{MON1-LEQ}: \text{mon-loc fg w1 }\subseteq \text{mon-loc fg w1 mon-env fg w1 }\subseteq \text{mon-env fg ww w1 by auto} \)

**From** \( \text{cil-ileq[OF LESPLIT(1)]} \) \text{mon-env-ileq[of map ENV w2 wfg fg]} \text{have MON2-LEQ: mon-ww fg w2 }\subseteq \text{mon-env fg w1 by simp} \)

**From** \( \text{LESPLIT(3) FS-FMT(3)} \) \text{ntrp-stack-decomp[of v]} \text{[\{\#\} w1 s' c1', simplified\} obtain \( v' rr \) where \text{DECOMP-LOC}: s' = v'\#rr@\text{\{\#\}} (\{w1.\{\#\}\},w1,(v'\#rr,c1'))\in trcl (ntrp fg) \) by (simp blast)

**From** \( \text{Cons.prems(2) LESPLIT(2)} \) \text{atUV U V ((\{\# s'\}+c1') + c2')} \text{by (simp add: union-ac)}

**Thus** \{case proof \{cases rule: atUV-union-cases\} \}

**Case left with** \textit{DECOMP-LOC(1) have ATUV: atUV U V (\{\# v'\#rr \#\}+c1') by simp} \}

**From** \( \text{Cons.hyps[OF - DECOMP-LOC(2) ATUV] cil-length[OF LESPLIT(1)\}] obtain Ml Me where IHAPP: (v, Ml, Me) }\in \) \text{RUV\{-cs fg U V Ml }\subseteq \text{mon-loc fg w1 Me }\subseteq \text{mon-env fg w1 by auto} \)

**From** \( \text{RUV-call[OF FS-FMT(2,4) S-ENTRY-PAT[of \{\#, simplified\} IHAPP(1)]} \)

103
have \((u, \text{mon fg} p \cup \text{mon-w} fg w \cup Ml, Me) \in \text{RUV-cs fg} U V\) using \(\text{IHAPP(3)}\) MON-PU MON1-LEQ by fastforce

moreover have \(\text{mon fg} p \cup \text{mon-w} fg w \cup Ml \subseteq \text{mon-loc} fg (\text{ce} \# \text{ww})\) using \(\text{FS-FMT(1)}\) \(\text{IHAPP(2)}\) MON1-LEQ by auto

moreover have \(Me \subseteq \text{mon-env} fg (\text{ce} \# \text{ww})\) using \(\text{IHAPP(3)}\) MON1-LEQ by auto

ultimately show \(\text{thesis}\) by blast

next

\(\text{case right}\) — Both nodes are reached from the spawned threads, we have to further distinguish whether both nodes are reached from the same thread or from different threads

then obtain \(s1' s2'\) where \(\text{R-STACKS:} \\{\#s1'\#\} + \{\#s2'\#\} \leq c2' \text{ atU-s U}\)

\(s1' \text{ atU-s V s2' by (unfold atUV-def) auto}\)

then obtain \(ce2'\) where \(\text{C2'FMT:} c2' = \{\#s1'\#\}+((\#s2'\#)+ce2')\) by (auto simp add: mset-le-exists-conv union-ac)

obtain \(q ceh w21 w22 ce21' ce22'\) where

\(\text{REVPSPLIT:} ch \!=\! \{\#[entry fg q]\#\} \!+\! ceh \{\#s2'\#\} + ce2' = ce21' + ce22' w2 \in w21 \otimes \text{an} fg w22\)

\(\text{mon fg q} \cap (\text{mon-c fg ceh} \cup \text{mon-ww fg w22}) = \{\}\)

\(\text{mon-c fg ceh} \cap (\text{mon-s fg ceh} \cup \text{mon-ww fg w21}) = \{\}\)

\((\{\#[entry fg q]\#\}, w21, \{\#s1'\#\} + ce21') \in \text{trcl (ntr fg)} (ceh, w22, ce22') \in \text{trcl (ntr fg)}\)

proof —

\(\text{case goal1}\)

from \(\text{ntr-reverse-split[of} ch w2 s1' \{\#s2'\#\} + ce2'\} \text{ntrp-valid-preserve-s[OF}\)

\(\text{SPLIT(1), simplified]} \text{C2'FMT} \text{LESPLIT(4)}\)

obtain \(seh ceh w21 w22 ce21' ce22'\) where

\(ch = \{\#seh\#\} + ceh \{\#s2'\#\} + ce2' = ce21' + ce22' w2 \in w21 \otimes \text{an} fg w22 \text{ mon-s}\)

\(\text{fg seh} \cap (\text{mon-c fg ceh} \cup \text{mon-ww fg w22}) = \{\}\)

\(\text{mon-c fg ceh} \cap (\text{mon-s fg seh} \cup \text{mon-ww fg w21}) = \{\}\)

\((\{\#seh\#\}, w21, \{\#s1'\#\} + ce21') \in \text{trcl (ntr fg)} (ceh, w22, ce22') \in \text{trcl (ntr fg)}\)

by (auto simp add: valid-unconc)

moreover from (this(1) CHFMT[of seh] obtain \(q\) where seh = [entry fg q] by auto

ultimately have \(ch = \{\#[entry fg q]\#\} + ceh \{\#s2'\#\} + ce2' = ce21' + ce22' w2 \in w21 \otimes \text{an} fg w22 \text{ mon fg q} \cap (\text{mon-c fg ceh} \cup \text{mon-ww fg w22}) = \{\}\)

\(\text{mon-c fg ceh} \cap (\text{mon-s fg ceh} \cup \text{mon-ww fg w21}) = \{\}\)

\((\{\#[entry fg q]\#\}, w21, \{\#s1'\#\} + ce21') \in \text{trcl (ntr fg)} (ceh, w22, ce22') \in \text{trcl (ntr fg)}\)

by auto

thus \(\text{thesis}\) using \(\text{goal1}\) by (blast)

qed

— For applying the induction hypothesis, it will be handy to have the reaching path in loc/env format:

from \(\text{ntrrs.hr2grtrp[where} c = \{\#\}, \text{ simplified, OF REVPSPLIT(6)}\) obtain \(sq'\)

\(\text{csp-q w2 w21}\) where

\(\text{R-CONV:} \{\#s1'\#\} + ce21' = \{\#sq'\#\} + \text{csp-q w2} w21 = \text{map le-rem-s w2 w21} ((\{\#[entry fg q]\}, \{\#\}), w2 w1, sq', csp-q) \in \text{trcl (ntr fg)}\) by blast

from \(\text{csp-ileq[OF REVPSPLIT(3)]}\) \(\text{mon-ww-ileq[of w2 w2 fg]}\) \(\text{mon-ww-ileq[of w2 w2 fg]}\) have \(\text{MON2N-LEQ:} \text{mon-ww fg w21} \subseteq \text{mon-ww fg w2}\)

\(\text{mon-ww fg w22} \subseteq \text{mon-ww fg w2 by auto}\)
from \textsc{REVSPLIT}(2) show \texttt{?thesis proof} (cases rule: \texttt{mset-anplum-dist-cases[case-names left' right']})

case \texttt{left' — Both nodes are reached from the same thread}

have \texttt{ATUV: atUV U V}\{(\#sq\#'\}+csp-q\}\texttt{using} right C2'\texttt{FMT R-STACKS(2,3)}
by (subst \texttt{R-CONV(1)[symmetric], subst left'(1)}) simp

from Cons.hyps[\texttt{OF - R-CONV(3) ATUV} cil-length[\texttt{OF \textsc{REVSPLIT}(3)}]
\texttt{cil-length[OF \texttt{LESPLIT(1)} R-CONV(2)] obtain Ml Me where IHAPP: (entry fg q, Ml, Me) \in RUV-cs fg U V Ml \subseteq mon-loc fg wwu21 Me \subseteq mon-env fg wwu21}
by auto

from \textsc{REVSPLIT}(1) S-ENTRY-PAT[of \{\#q\}', simplified] have S-ENTRY: (v, mon-w fg w, \{\#q\}') \in S-cs fg I by simp

have \texttt{MON-COND: (mon-n fg u \cup mon fg p) \cap (Ml \cup Me) = \{\}} proof —

from \texttt{R-CONV(2)} have mon-w fg w21 = mon-loc fg wwu21 \cup mon-env fg wwu21
by (simp add: mon-ww-of-le-rem)

with IHAPP(2,3) \texttt{MON2N-LEQ(1) MON-PU MON2-LEQ} show \texttt{?thesis by blast}

next

case \texttt{right' — The nodes are reached from different threads}

from \texttt{R-STACKS(2,3)} have \texttt{ATUV: atUV U} ((\#sq\#'\}+csp-q) atUV V ce22'
by (\texttt{-- (subst \texttt{R-CONV(1)[symmetric], simp, subst right'(1), simp})})

— We have to reverse-split the second path again, to extract the second interesting thread

obtain q'\# w22' ce22e' where \textsc{REVSPLIT'}: [\texttt{entry fg q'}\# :\# ceh w22'\#ce22 ce22e' \leq \# ce22' atUV V ce22e'(\{\#[\texttt{entry fg q'}\#\}w22',ce22e')\in trcl (ntrp fg)] proof

— case goal1

from ntr-reverse-split-atU[\texttt{OF - ATUV(2) \textsc{REVSPLIT}(7)}] ntrp-valid-preserve-s[\texttt{OF SPLIT(1), simplified}] \textsc{REVSPLIT(1)} obtain sq'' w22' ce22e' where

sq'' :\# ceh w22'\#\leq w22 ce22e' \leq \# ce22' atUV V ce22e'(\{\#sq''\#\},w22',ce22e')\in trcl (ntrp fg) by (auto simp add: valid-unconc)

— moreover from CHFMT[of sq''] \textsc{REVSPLIT(1)} this(1) obtain q' where

sq''=[\texttt{entry fg q'}] by auto

ultimately show \texttt{?thesis using goal1 by blast}

qed

from ntrrs,gr2grp[wce\{\#\}, simplified, \texttt{OF \textsc{REVSPLIT'}(5)}] obtain sq'' ce22e' wwu22' where \texttt{R-CONV'}: ce22e' = \{\#sq''\#\}+ce22e' wwu22'\= map le-rem-s wwu22'
((\texttt{entry fg q'}\#,\{\#\},wwu22',(sq''\#ce22e'))\in trcl (ntrp fg) by blast

— From the soundness of the RU-constraint system, we get the corresponding entries

105
from RU-sound[OF R-CONV(3) ATUV(1)] obtain MI Me h where RU: (entry fg q, MI, Me, h) ∈ RU-cs fg U MI ⊆ mon-loc fg wW21 Me ⊆ mon-env fg wW21 h ≤ oah (map (cml fg) wW21) by blast

from RU-sound[OF R-CONV’(3), of V] REVSPLIT’(4) R-CONV’(1) obtain MI’ Me’ h’ where RV: (entry fg q’, MI’, Me’, h’) ∈ RU-cs fg V MI’ ⊆ mon-loc fg wW22’ Me’ ⊆ mon-env fg wW22’ h’ ≤ oah (map (cml fg) wW22’) by auto

from S-ENTRY-PAT[of {#q#} + {#q’#}, simplified] REVSPLIT’(1) REVSPLIT’(1) have S-ENTRY: (v, mon-w fg w, {#q#} + {#q’#}) ∈ S-cs fg (2::nat) by (simp add: numerals)

have (u, mon fg p ∪ mon-w fg w, MI ∪ Me ∪ MI’ ∪ Me’) ∈ RUV-cs fg U V proof (rule RUV-split-ce[OF FS-FMT(2,4) S-ENTRY - RU(1) RV(1)])

from MON-PU MON2-LEQ MON2N-LEQ RU(2,3) MON2N-LEQ R-CONV(2) R-CONV’(2) mon-ww-ileq[OF REVSPLIT’(2), of fg] RU(2,3) RV(2,3) show (mon-n fg u ∪ mon fg p) ∩ (MI ∪ Me ∪ MI’ ∪ Me’) = {} by (simp add: mon-ww-of-le-rem) blast

next

from ah-interleaveable1[OF REVSPLIT’(3)] have oah (map (cml fg) wW21) by (rule oah-ileq)

...
∈ RU-cs fg V Ml' ⊆ mon-ww fg w2 Me' ⊆ mon-ww fg w2 h' ≤ αah (map (on fg) w2) proof —
  case goal1
  — We have to extract the interesting thread from the spawned threads in order to get an entry in RU fg V
  obtain q' w2' c2i' where REV_SPLIT: [entry fg q'] :# ch w2' ≤ w2 c2i' ≤ # c2' at U V c2i' ({# | entry fg q' #}, w2' ∈ c2i') ∈ trcl (ntrp fg)
  using ntr-reverse-split-atU [OF - br(2)] REV_SPLIT(4) ntrp-valid-preserve-s[OF SPLIT(1), simplified] CH_FMT by (simp add: valid-unc onc) blast
  from ntrs.trgr fratp [where c=#] simplified, OF REV_SPLIT(5) obtain s2i' c2ie' w2' where R_CONV: c2i' = #{s2i' #} + c2ie' w2' = map le-rem-s w2' (((entry fg q'), #), w2', s2i', c2ie') ∈ trcl (ntrp fg).
  from RU-sound[OF R_CONV(3), of V] REV_SPLIT(4) R_CONV(1) obtain MI' Me' h' where RV: (entry fg q', MI', Me', h') ∈ RU-cs fg V MI' ⊆ mon-loc fg w2' Me' ∈ mon-env fg w2' h' ≤ αah (map (onl fg) w2') by auto
  moreover have mon-loc fg w2' ⊆ mon-ww fg w2 mon-env fg w2' ⊆ mon-ww fg w2 using mon-ww-ileq [OF REV_SPLIT(2), of fg] R_CONV(1) by (auto simp add: mon-ww-of-le-s simp del: map-map intro: αah-ileq del: predicate2I)
  ultimately show ?thesis using goal1 REV_SPLIT(1) by (blast intro: order-trans)
qed

from S-ENTRY-PAT[of {#|q|}, simplified] RV(1) have S-ENTRY: (v, mon-w fg w, {# | q |}) ∈ S-cs fg # by simp
  have (u, mon fg p u mon-w fg w ∪ ML, Me ∪ MI' ∪ Me') ∈ RU-cs fg U V proof (rule RU-split-le[OF FS_FMT(2,4) S-ENTRY - RU(1) LV(2)])
  from MON-PU MONI-LEQ MON2-LEQ RV(3) RV(3,4) show (mon-n fg u u mon fg) ∩ (Me ∪ MI' ∪ Me') = {} by blast
next
  from ah-interleavable1[OF LESPLIT(1)] have αah (map (onl fg) w1) [*] αah (map (onl fg) w2) by simp
  thus h [*] h' using RU(4) RV(5) by (auto elim: ah-leq-il)
qed (simp)

moreover have mon fg p u mon-w fg w ∪ ML ⊆ mon-loc fg (ee # ww) using FS_FMT(1) MONI-LEQ RU(2) by (simp) blast
moreover have Me ∪ MI' ∪ Me' ⊆ mon-env fg (ee # ww) using MONI-LEQ MON2-LEQ RU(3) RV(3,4) by (simp) blast
ultimately show ?thesis by blast
next
  case rl — The second node is reached from the local thread, the first one from a spawned thread. This case is symmetric to the previous one
  from RU-sound[OF DECOMP-LOC(2), of V] rl(1) DECOMP-LOC(1) obtain MI Me h where RV: (v, MI, Me, h) ∈ RU-cs fg V MI ⊆ mon-loc fg w1 Me ⊆ mon-env fg w1 h ≤ αah (map (onl fg) w1) by auto
  obtain MI' Me' h' q' where RU: (entry fg q') :# ch (entry fg q', MI', Me', h') ∈ RU-cs fg U MI' ⊆ mon-ww fg w2 Me' ⊆ mon-ww fg w2 h' ≤ αah (map (onl fg) w2) proof —
  case goal1

107
— We have to extract the interesting thread from the spawned threads in order to get an entry in RU fg V

obtain q' w2' c2i' where REVSPILT: [entry fg q'] :: ch w2'≤w2 c2i' ≤# c2' atUV U c2'( ([#entry fg q']#, w2', c2i') ∈ trcl (ntrp fg))

using nfr-reverse-split-atUV (OF- ri (2) LESPLIT (4)) ntrp-valid-preserve-s [OF SPLIT (1), simplified] CHFMT by (simp add: valid-unconc) blast

from ntr-sgr2fgtmap (where c= [#], simplified, OF REVSPILTR (5)) obtain s2i' c2ie' ww2' where - R-CONV: c2i'= {#s2i' #} + c2ie' w2' = map le-rem-s ww2' (((entry fg q'), {#}), w2', s2i', c2ie') ∈ trcl (ntrp fg).

from RU-sound (OF R-CONV (3), of U) REVSPILTR (4) R-CONV (1) obtain MI' Me' h' where RU: (entry fg q', MI', Me', h') ∈ RU-cs fg U MI' ⊆ mon-loc fg ww2' Me' ⊆ mon-env fg ww2' h' ≤ αah (map (onl fg) ww2') by auto

moreover have mon-loc fg ww2' ⊆ mon-ww fg w2 mon-env fg ww2' ⊆ mon-ww fg w2 using mon-ww-leq (OF REVSPILTR (2), of fg) R-CONV (2) by (auto simp add: mon-ww-of-le-rem)

moreover have αah (map (onl fg) ww2') ≤ αah (map (onl fg) w2) using REVSPILTR (2) R-CONV (2) by (auto simp add: onl-onl[symmetric] le-list-map map-map[symmetric] simp del: map-map intro: αah-leq del: predicate2I)

ultimately show ?thesis using goal1 REVSPILTR (1) by (blast intro: order-trans) qed

from S-ENTRY-PAT[of {#q' #}, simplified] RU (1) have S-ENTRY: (v, mon-w fg w, {#q' #}) ∈ S-cs fg I by simp

have (u, mon fg p ∪ mon-w fg w ∪ MI, Me ∪ MI' ∪ Me') ∈ RU-cs fg U V proof (rule RUV-split-cl (OF FS-FMT (2, 4) S-ENTRY - RV (1) RU (2)))

from MON-PU MON1-LEQ MON2-LEQ RV (3) RU (3, 4) show (mon-n fg u ∪ mon fg p) ∩ (Me ∪ MI' ∪ Me') = {} by blast

next

from ah-interleaveable1 [OF LESPLIT (1)] have αah (map (onl fg) w1) [s] αah (map (onl fg) w2) by simp

thus h [s] h' using RV (4) RU (5) by (auto elim: ah-leq-il) qed (simp)

moreover have mon fg p ∪ mon-w fg w ∪ MI ⊆ mon-loc fg (ee # wv) using FS-FMT (1) MON1-LEQ RV (2) by (simp) blast

moreover have Me ∪ MI' ∪ Me' ⊆ mon-env fg (ee # wv) using MON1-LEQ MON2-LEQ RV (3) RU (3, 4) by (simp) blast

ultimately show ?thesis by blast qed

lemma (in flowgraph) RUV-precise: (u, MI, Me) ∈ RU-cs fg U V

|⇒ w s' c'

(|{w}, {#}), w, (s', c') ∈ trcl (ntrp fg) ∧ atUV U V (|#s' #| + c') ∧ mon-loc fg w = MI ∧
mon-env fg w = Me

proof (induct rule: RUV-cs.induct)

case (RVU-call u p u' v M P MI Me) then obtain ww s' c' where IH: ((|w|, {#}), ww, s', c') ∈ trcl (ntrp fg) atUV U V (|#s' #| + c') mon-loc fg ww = MI mon-env fg ww = Me by blast

108
from $S$-precise-ntrp[OF RUV-call(3,2,1), simplified] obtain $w$ $ch$ where $FS$: $((\{u\}, \{\#\}), LOC (LCall p \# w), [v, u]', ch) \in ntrp fg P = \{\#\} M = mon-w fg w$ $mon-n fg v = mon fg p mon-c fg ch = \{}$ by blast

note $FS(1)$
also have $(((v, u)', ch), w w, s' @ [u]', c' + ch) \in trcl (ntrp fg)$
using ntrp-add-context[OF ntrp-stack-comp[OF IH(1), of [u'], of ch, simplified] FS(5) IH(4) RUV-call.hyps(6) mon-n-same-proc[OF edges-part[OF RUV-call.hyps(1)]]]
by simp
finally have $(((\{u\}, \{\#\}), LOC (LCall p \# w) \# w w, s' @ [u]', c' + ch) \in trcl (ntrp fg)$.
moreover from IH(2) have $atUV U V ((\#s' @ [u']\#)+ (c' + ch))$ by auto
moreover have $mon-loc fg (LOC (LCall p \# w) \# w w) = mon fg p \cup M \cup \{\#s'\#\} + che$ by auto
ultimately show $?case$ by blast

next
case $(RU V$-spawn $u p u'$ $v M P q M l M e)$ then obtain $w w s' c'$ where $IH$: $(((\{entry fg q\}, \{\#\}), w w, s', c') \in trcl (ntrp fg)$ $atUV U V ((\#s'\#) + c') mon-loc fg w w = M l mon-loc fg v = M e$ by blast
from $S$-precise-ntrp[OF RUV-spawn(3,2,1), simplified] mset-size1elem[OF OF - RUV-spawn(4)]
obtain $w$ $ch$ where $FS$: $(((\{u\}, \{\#\}), LOC (LCall p \# w), [v, u'], \{\#entry fg q\#\} + che) \in ntrp fg P=\{\#q\#\} M = mon-w fg w mon-n fg v = mon fg p mon-c fg \{(\#entry fg q\#) + che\} = \{}$ by (auto simp add: mon-c-unconc mon-ww-of-le-rem)
moreover have $(((\{v, u\}', che + \{\#entry fg q\#\}), map ENV (map le-rem-s w w), ([v, u'], che+\{(\#s'\#) + che\})) \in trcl (ntrp fg)$
using ntrp2ntrp[OF gtrp2gtr[OF OF IH(1)], of [v,u'] che] IH(3,4) RUV-spawn(7)
$FS(4,5) mon-n-same-proc[OF OF edges-part[OF RUV-spawn(1)]]$
by (auto simp add: mon-c-unconc mon-ww-of-le-rem)
ultimately have $(((\{u\}, \{\#\}), LOC (LCall p \# w) \# map ENV (map le-rem-s w w), ([v, u'], che+\{(\#s'\#) + c\})) \in trcl (ntrp fg)$
by (auto simp add: union-ac)
moreover have $atUV U V ((\{\#v, u'\#\}) + (che+\{(\#s'\#) + c\}))$ using $IH(2)$
ultimately show $?case$ by blast

next
case $(R UV$-split-le $u p u'$ $v M P q M l M e h M l' M e' h')$
— Get paths from precision results
from $S$-precise-ntrp[OF RUV-split-le(3,2,1), simplified] mset-size1elem[OF OF - RUV-split-le(4)]
obtain $w$ $ch$ where $FS$: $(((\{u\}, \{\#\}), LOC (LCall p \# w), [v, u'], \{\#entry fg q\#\} + che) \in ntrp fg P=\{\#q\#\} M = mon-w fg w mon-n fg v = mon fg p mon-c fg \{(\#entry fg q\#) + che\} = \{}$ by (auto elim: mset-le-addE)
from RU-precise[OF RUV-split-le(5)] obtain \(ww1 \ s1' \ c1'\) where \(P1:\ ([v],\ \{\#\}, \ \text{ww1}, \ s1', \ c1') \in \text{trcl (ntrp fg) atU U \ (#s1'\#) + c1'}\) mon-loc fg \(ww1 = Ml \ \text{mon-env fg} \ \text{ww1} = Me' \ \alphaah (\text{map (envl fg) \ \text{ww1}}) = h'\) by blast

from RU-precise[OF RUV-split-le(6)] obtain \(ww2 \ s2' \ c2'\) where \(P2:\ ([\{entry fg q\}, \ \{\#\}], \ \text{ww2}, s2', c2') \in \text{trcl (ntrp fg) atU V \ (#s2'\#) + c2'}\) mon-loc fg \(ww2 = Ml' \ \text{mon-env fg} \ \text{ww2} = Me' \ \alphaah (\text{map (envl fg) \ \text{ww2}}) = h'\) by blast

— Get combined path from the acquisition history interleavability, need to remap loc/env-steps in second path

from \(P2(5)\) have \(\alphaah (\text{map (envl fg) \ (map le-rem-s \ \text{ww2})}) = h'\) by (simp add: envl o-assoc)

with \(P1(5)\) RUV-split-le(8) obtain \(ww\) where \(IL: \ \text{ww} \in \text{ww1} \odot \alphaah \ (\text{map ENV (map le-rem-s \ \text{ww2})})\) using ah-interleavable2 by (force)

— Use the ntrp-unsplit-theorem to combine the executions

from ntrp-unsplit[where ca={\#}, OF IL P1(1) gtrp2gtr[OF P2(1)], simplified] have \([\{v\}, \ \{\{\text{entry fg q}\}\#\}]\), \(ww, s1', c1' + \ (#s2'\#) + c2'\) \in \text{trcl (ntrp fg) using FS(4.5) RUV-split-le(7)}

by (auto simp add: mon-c-unconc mon-ww-of-le-rem P2(3,4))

from ntrp-add-context[OF ntrp-stack-comp[OF this, of [\[u'\]], of che] have \(\{v\} \ \\text{\atU [u']}\}, \ \{\{\text{entry fg q}\}\#\} + \text{che}\), \(ww, s1' @ [u'], \ c1' + \ (#s2'\#) + c2' + \text{che}\) \in \text{trcl (ntrp fg)}


with FS(1) have \(\{[u], \ \{\#\}\}, \ \text{LOC (LCall p \ # w)} \ # \ \text{ww}, (s1' @ [u'], \ c1' + \ (#s2'\#) + c2' + \text{che}) \in \text{trcl (ntrp fg)}\) by simp

moreover have atU V [\{#s1'\# @ [u']\#\}+(c1' + \ (#s2'\#) + c2' + \text{che})]

using P1(2) P2(2) by auto

moreover have mon-loc fg (LOC (LCall p \ # w) \ # \ \text{ww}) = mon fg p \ \text{M} \ \text{Ml} using FS(3) P1(3) mon-loc-cil[OF IL, of fg] by (auto simp del: map-map)

moreover have mon-env fg (LOC (LCall p \ # w) \ # \ \text{ww}) = Me' \ \text{Ml'} using P1(4) P2(3,4) mon-env-cil[OF IL, of fg] by (auto simp add: mon-ww-of-le-rem simp del: map-map)

ultimately show ?case by blast

next

case (RU-precise-\text{el} \ \text{u p u' v M P q Ml Me h Ml'} Me' h') — This is the symmetric case to RUV-split-le, it is proved completely analogously, just need to swap \(U\) and \(V\).

— Get paths from precision results

from S-precise-ntrp[OF RUV-split-el(3,2,1), simplified] mset-size1elem[OF RUV-split-el(4)] obtain \(ww\ \text{\che}\) where

\(FS: (([u], \ \{\#\}), \ \text{LOC (LCall p \ # w)} \ [v, u'], \ \{\{\text{entry fg q}\}\#\} + \text{che}) \in \text{ntrp fg} \ P=[\#q\#\}] \ \text{M = mon-w fg w mon-n fg v = mon fg p mon-c fg (\{#entry fg q\}\# +che) = \}}\) by (auto elim: mset-le-addE)

from RU-precise[OF RUV-split-el(5)] obtain \(ww1 \ s1' \ c1'\) where \(P1:\ ([v], \ \{\#\}), \ \text{ww1}, s1', c1') \in \text{trcl (ntrp fg) atU U \ (#s1'\#) + c1'}\) mon-loc fg \(ww1 = Ml \ \text{mon-env fg} \ \text{ww1} = Me' \ \alphaah (\text{map (envl fg) \ \text{ww1}}) = h'\) by blast

from RU-precise[OF RUV-split-el(6)] obtain \(ww2 s2' c2'\) where \(P2:\ ([\{entry fg q\}, \ \{\#\}], \ \text{ww2}, s2', c2') \in \text{trcl (ntrp fg) atU U \ (#s2'\#) + c2'}\) mon-loc fg \(ww2 = Ml' \ \text{mon-env fg} \ \text{ww2} = Me' \ \alphaah (\text{map (envl fg) \ \text{ww2}}) = h'\) by blast
— Get combined path from the acquisition history interleavability, need to remap loc/env-steps in second path

from $P2(5)$ have $a_{\text{eh}} \cdot \text{(map (\alpha n \text{fg}) (map \text{ENV} (\text{map le-rem-s w}_{2}))}) = h' \text{ by (simp add: an-\text{cel} o-assoc)}$

with $P1(5)$ RUV-split-el(8) obtain $w_w$ where $\text{IL} \colon w_w\in w_w1 \otimes _{\text{an \text{fg}}} (\text{map \text{ENV}} (\text{map le-rem-s w}_{2}))$ using ah-interleaveable2 by (force)

— Use the ntr-unsplit-theorem to combine the executions

from ntr-unsplit[where ca={#}, OF IL P1(1) getp2grOF OF P2(1)], simplified] have $([[w_w], \{#|\text{entry fg q}|\}])$, $w_w$, $s_{1'}$, $c_{1'}' + (\{#s_{2'}|\} + c_{2'}') \in \text{trcl (ntrp fg)}$ using $FS(4,5)$ RUV-split-el(7)

by (auto simp add: mon-c-unconc mon-ww-of-le-rem $P2(3,4)$)

from ntrp-add-context[OF ntrp-stack-comp[OF this, of [u'], of che] have $([[w] @ [u'], \{#|\text{entry fg q}|\} + \text{che})$, $w_w$, $s_{1'} @ [u']$, $c_{1'} + (\{#s_{2'}|\} + c_{2'}) + \text{che}) \in \text{trcl (ntrp fg)}$


with $FS(1)$ have $([[u], \{#\})$, LOC $\langle \text{LCall p \# w} \rangle$ $\not\in w_w$, $(s_{1'} @ [u']$, $c_{1'} + (\{#s_{2'}|\} + c_{2'}) + \text{che}) \in \text{trcl (ntrp fg)}$ by simp

moreover have atUV $U V$ $((\{#s_{1'}| @ [u']})) + (c_{1'} + (\{#s_{2'}|\} + c_{2'}) + \text{che})$
have $P1(2)$ $P2(2)$ by auto

moreover have mon-loc $\text{fg}$ (LOC $\langle \text{LCall p \# w} \rangle$ $\not\in w_w$) = mon $\text{fg}$ $\cup M \cup M'l$ using $FS(3)$ $P1(3)$ mon-loc-cil[OF IL, of fg] by (auto simp del: map-map)

moreover have mon-\text{env} $\text{fg}$ (LOC $\langle \text{LCall p \# w} \rangle$ $\not\in w_w$) = $M \cup M'l \cup M' \text{'}$ using $P1(4)$ $P2(3,4)$ mon-env-cil[OF IL, of fg] by (auto simp add: mon-ww-of-le-rem simp del: map-map)

ultimately show ?case by blast

next

case (RU V-split-ee $u$ $u' v$ $M P q q'$ $M'l M h M' M'h$)

— Get paths from precision results

from $S$-precise-ntrp[OF RUV-split-ee(3,2,1), simplified] mset-size2elem[OF - RUV-split-ee(4)] obtain $w_w$ $c_{1'}$ where $FS$: $([[u], \{#\})$, LOC $\langle \text{LCall p \# w} \rangle$, $[v, u']$, $\{#|\text{entry fg q}|\} + \{#|\text{entry fg q'}|\} + \{#|\text{entry fg q'|}\} + \text{che}) \in \text{ntrp fg}$ $P = \{#q|\} + \{#q'|\} M = \text{mon-\text{fg}} w_w \text{ mon-n fg v} = \text{mon-\text{fg}}$ $\text{p} \text{ mon-c fg} (\{#|\text{entry fg q}|\} + \{#|\text{entry fg q'}|\} + \text{che}) = \{\}

by (auto elim: mset-\text{addE})

from RU-precise[OF RUV-split-ee(5)] obtain $w_w1$ $s_{1'}$ $c_{1'}$ where $P1$: $([[\text{entry fg q'}], \{#\})$, $w_w1$, $s_{1'}$, $c_{1'}') \in \text{trcl (ntrp fg)}$ atUV $U$ $((\{#s_{1'}| \} + c_{1'})$ mon-loc $fg$ $w_w1 = M l$ mon-\text{env} $fg w_w1 = N e a h$ (map (\alpha n \text{fg}) $w_w1) = h$ by blast

from RU-precise[OF RUV-split-ee(6)] obtain $w_w2$ $s_{2'}$ $c_{2'}$ where $P2$: $([[\text{entry fg q'}], \{#\})$, $w_w2$, $s_{2'}$, $c_{2'}') \in \text{trcl (ntrp fg)}$ atUV $V$ $((\{#s_{2'}| \} + c_{2'})$ mon-loc $fg$ $w_w2 = M' l$ mon-\text{env} $fg w_w2 = N e a h$ (map (\alpha n \text{fg}) $w_w2) = h'$ by blast

— Get interleaved paths, project away loc/env information first

from $P1(5)$ $P2(5)$ have $a_{\text{eh}} \cdot \text{(map (\alpha n \text{fg}) (map \text{le-rem-s w}_{1}))} = h \alpha a h$ (map (\alpha n \text{fg}) (map \text{le-rem-s w}_{2})) = $h' \text{ by (auto simp add: an-\text{cel} o-assoc)}$

with $RU V$-split-el(8) obtain $w_w$ where $\text{IL} : w_w \in (\text{map \text{le-rem-s w}_{1}) \otimes _{\text{an \text{fg}}} (\text{map \text{le-rem-s w}_{2})}$ using ah-interleaveable2 by (force simp del: map-map)

— Use the ntr-unsplit-theorem to combine the executions
from ntr-unsplit[OF IL gtrp2gtr[OF P1(1)] gtrp2gtr[OF P2(1)], simplified] have PC: {(entry fg q)#} + {entry fg q'#} ∈ trcl (ntr fg) using FS(5) by (auto simp add: mon-c-unconc)

Moreover have \((\text{entry fg q}) + (\text{entry fg q}'\})\) ∈ trcl (ntr fg)

using RUV-split-ce(7) FS(5) mon-ww-cil[OF IL, of fg] FS(4) mon-n-same-proc[OF edges-part[OF RUV-split-ce(1)]]) by (auto simp add: mon-c-unconc mon-ww-of-le-rem P1(3,4) P2(3,4))

with FS(1) have \((\text{entry fg q}) + (\text{entry fg q}'\})\) ∈ trcl (ntr fg)

using P1(2) P2(2) by auto

moreover have mon-loc fg (LOC (LCall p # w) # map ENV ww) = mon fg p ∪ M using FS(3) by auto

moreover have mon-env fg (LOC (LCall p # w) # map ENV ww) = Ml ∪ Me ∪ Ml' ∪ Me' using mon-ww-cil[OF IL, of fg] by (auto simp add: P1(3,4) P2(3,4) mon-ww-of-le-rem)

ultimately show ?case by blast

qed

end

10 Main Result

theory MainResult
imports ConstraintSystems
begin

At this point everything is available to prove the main result of this project: The constraint system RUV-cs precisely characterizes simultaneously reachable control nodes w.r.t. to our semantic reference point.

The „trusted base” of this proof, that are all definitions a reader that trusts the Isabelle prover must additionally trust, is the following:

- The flowgraph and the assumptions made on it in the flowgraph- and eflowgraph-locales. Note that we show in Section 6.4 that there is at least one non-trivial model of eflowgraph.
- The reference point semantics (refpoint) and the transitive closure operator (trcl).
- The definition of atUV.
- All dependencies of the above definitions in the Isabelle standard libraries.
proof

— The proof uses the soundness and precision theorems wrt. to normalized paths
(flowgraph.RUV-sound, flowgraph.RUV-precise) as well as the normalization result,
i.e. that every reachable configuration is also reachable using a normalized path
(flowgraph.normalize) and, vice versa, that every normalized path is also a usual
(path (ntr-is-br)). Finally the conversion between our working semantics and the
semantic reference point is exploited (flowgraph.refpoint-eq).

proof

case goal1 then obtain w c' where C: ({#|entry fg (main fg)|#}, w, c') ∈ trcl
(tr fg) atUV U V c' by (auto simp add: refpoint-eq)
  from normalize[OF C(1), of main fg, simplified] obtain ww where
  ({#|entry fg (main fg)|#}, ww, c') ∈ trcl (ntr fg) by blast
  from ntrs.gtr2gtrp[where c={#}, simplified, OF this] obtain s' ce' wwl where
  1: c'={#|s'|#}+uc' ww = map le-rem-s wwl
  ({#|entry fg (main fg)|#}, {#}), wwl, s', ce') ∈ trcl (ntrp fg) by blast
  with C(2) have 2: atUV U V {{#|s'|#}+ce'} by auto
  from RUV-sound[OF 1(3) 2] show ∃ Ml Me. (entry fg (main fg), Ml, Me) ∈
  RUV-CS fg U V by blast

next

case goal2 then obtain Ml Me where C: (entry fg (main fg), Ml, Me) ∈
  RUV-CS fg U V by blast
  from RUV-precise[OF C] obtain wwl s' c' where P: 
  ({#|entry fg (main fg)|#}, wwl, s', c') ∈ trcl (ntrp fg) atUV U V
  {{#|s'|#}+c'} by blast
  from gtrp2gtr[OF P(1)] have
  ({#|entry fg (main fg)|#}, map le-rem-s wwl,
  {{#|s'|#}+c'}) ∈ trcl (ntrp fg) by (auto)
  from ntr-is-br[OF this] P(2) have
  ∃ w c'. ({#|entry fg (main fg)|#}, w, c') ∈
  trcl (tr fg) ∧ atUV U V c' by blast
  thus ∃ w c'. ({#|entry fg (main fg)|#}, w, c') ∈ trcl (refpoint fg) ∧ atUV U V
  c' by (simp add: refpoint-eq)

qed

end

11 Conclusion

We have formalized a flowgraph-based model for programs with recursive
procedure calls, dynamic thread creation and reentrant monitors and its
operational semantics. Based on the operational semantics, we defined a
conflict as being able to simultaneously reach two control points from two
given sets U and V when starting at the initial program configuration, just
consisting of a single thread at the entry point of the main procedure. We
then formalized a constraint-system-based analysis for conflicts and proved
it sound and precise w.r.t. the operational definition of a conflict. The main
idea of the analysis was to restrict the possible schedules of a program. On
the one hand, this restriction enabled the constraint system based analysis, on the other hand it did not change the set of reachable configurations (and thus the set of conflicts).

We characterized the constraint systems as inductive sets. While we did not derive an executable algorithm explicitly, the steps from the inductive sets characterization to an algorithm follow the path common in program analysis and pose no particular difficulty. The algorithm would have to construct a constraint system (system of inequalities over a finite height lattice) from a given program corresponding to the inductively defined sets studied here and then determine its least solution, e.g. by a worklist algorithm. In order to make the algorithm executable, we would have to introduce finiteness assumptions for our programs. The derivation of executable algorithms is currently in preparation.

A formal analysis of the algorithmic complexity of the problem will be presented elsewhere. Here we only present some results: Already the problem of deciding the reachability of a single control node is NP-hard, as can be shown by a simple reduction from SAT. On the other hand, we can decide simultaneous reachability in nondeterministic polynomial time in the program size, where the number of random bits depends on the possible nesting depth of the monitors. This can be shown by analyzing the constraint systems.

Acknowledgement We thank Dejvuth Suwimonteerabuth for an interesting discussion about static analysis of programs with locks. We also thank the people on the Isabelle mailing list for quick and useful responses.

References


